Reflective Transaction Processing – Motivations and Issues

Weihai Yu

Department of Computer Science
University of Tromsø

weihai@cs.uit.no

Abstract

In this paper I will discusses *reflective transaction processing*, a new research area that I believe has its potential. I start with a motivating example, i.e., using reflection as a mechanism to address the heterogeneity issues in distributed transaction processing. Other usages of reflection in transaction processing will also be briefly mentioned. I will then explain what reflection is and discuss the related issues on reflective transaction processing. As the discussion goes on, it will become clear that reflective transaction processing itself can be more general and difficult than the heterogeneity issues. However, *reflective transaction management*, a sub-area of reflective transaction processing, can be more realistic as a shorter-term research target.

1 Distributed transactions in heterogeneous environments – a hard problem

Distributed applications at or beyond enterprise scale are getting increasingly important. Transaction processing plays a key role in these applications where consistency is a primary requirement. These applications run typically in heterogeneous environments that involve database systems from different vendors and subsystems (or application entities, which again may involve multiple database systems) developed independently in different projects.

Important with such distributed applications is the support for *interoperability* among involved database systems and application entities. Interoperability stands for the ability of two or more software entities to cooperate despite of differences in language, interface and execution platform [15].

Middleware provides a means of interoperability among heterogeneous application entities or between applications and database systems. More specifically, middleware provides platform and/or programming language independence. An entity (client) can invoke another one (server) regardless of the platform on which they are running and/or the programming language in which they are implemented.

However, supporting distributed transactions in a general heterogeneous environment is impossible today. All involved entities must use the same transaction processing system, otherwise, global transactional properties will not be guaranteed.

Interoperability issues can be explained by the plug-and-socket metaphor [15]: Plug-compatibility arises most literally with electrical appliances that require both static compatibility of shape and dynamic compatibility of voltage and frequency. The client-server software paradigm is a plug-and-socket paradigm with static compatibility

(interoperability) specified by types and dynamic compatibility (interoperability) by protocols. Middleware provides static interoperability through standardized interface definition languages. Distributed transaction processing has, in addition to static interoperability issues, dynamic interoperability issues to be addressed.

Distributed transaction processing consists of three parts: concurrency control, recovery control and distributed transaction management.

- Concurrency control assures correct behavior of concurrent transactions through schedulers.
- Recovery control protects data in face of system failures through logging.
- Distributed transaction management coordinates recovery and control activities among different sites.

All these three parts can be sources of heterogeneity:

- Concurrency control schemes.

  There are pessimistic versus optimistic schemes, and lock-based versus timestamp-ordering, etc. Even for lock-based schemes, there are variations like two-phase locking, strict two-phase locking, altruistic locking, lock conflict according to read/write or general method semantics of abstract data types, and so on. Some of the schemes may work together; some may not.

- Recovery mechanisms.

  Different recovery mechanisms include undo, redo, redo/undo, combined by various savepoint and checkpoint mechanisms. Again, some of them can be combined; some of them may not.

- Transaction management.

  This keeps track of transaction context information and coordinates concurrency control and recovery activities. Typical tasks include transaction enlistment and commitment processing. Particularly for transaction commitment, there are one-phase commit, two-phase commit and three-phase commit protocols. Even for the popular two-phase commit protocol, there are variations like presumed-commit, presumed-abort, implicit-yes-vote, transfer-of-commit, etc. Again, combining them is a very difficult task.

Transaction processing standards (such as X/Open [17] and OMG [14]) help solve some of the problems regarding static interoperability. For example, when the involved systems support XA interfaces, concurrency control and recovery are then to a large extent local issues. Transaction management remains the issue. *Autonomy* makes this issue even harder. Informally, autonomy of a local transaction subsystem (can be distributed or non-distributed) represents the ability of the transaction processing subsystem to execute events without any curtailment. More specifically, this means that the transaction processing subsystem, when participated in a larger distributed environment, will not be either forced to or prevented from executing events that otherwise are not forced to or prevented from by the local system [4]. For example, if a local system does not provide (or publish) the `prepare-to-commit()` event, forcing it to execute the event violates its autonomy. Similarly, preventing a local system from aborting a transaction (for example, because it is in the uncertainty status without its awareness) also violates its autonomy. When some extended transaction models are used, local transaction managers may be required to support additional operations, such

as the `split()` operation of the split-join transaction model (see [6]). If a local system is enforced to execute this operation, which it originally does not provide, autonomy is violated. Obviously, various requirements on autonomy hinder local systems from collaborating with each other.

It is interesting to observe that certain non-functional features like security may have impact on the degree of autonomy with respect to transaction processing. Notice that autonomy is determined by whether certain operations like `prepare-to-commit()` can be externalized and whether new operations like `split()` can be added. An important factor of deciding these is whether the transaction initiator (application level) or the current coordinator (system level) is trusted. Thus, autonomy of a database server may be dependent on the current run-time context of a transaction inclusive other seemingly unrelated features.

2 Reflective transaction processing – a possible solution

Solutions, though, exist for specific problems due to particular sources of heterogeneity in particular situations. Some of them require application specific knowledge of the transactions. Let us take the problem of autonomous and closed resource manager as an example. A resource manager is *closed* if it does not externalize the `prepare-to-commit()` operation for two-phase commitment. For the discussions later, we list some of the possible solutions:

1. Compensation [7] is often applicable to part of a transaction that involves a closed resource manager. The sub-transaction at a local system can commit as an independent top transaction without participating global commitment processing. If necessary, a compensation transaction can be executed to logically undo the committed effects.
2. As a very special exception, the resource manager does not have to be open if it happens to be located at the same site of the coordinator of the transaction.
3. If the transaction is read-only, some lower degree of consistence (in terms of lower isolation levels) is often acceptable. Read locks can be released as soon as data have been read, without waiting until the entire transaction terminates.
4. If the current transaction model supports alternative transactions (see [6]), it would be possible to spawn an alternative transaction instead of running the transaction on the closed resource manager.
5. Another possible solution is to split the transaction into smaller ones such that each of these new smaller transactions only involves one closed resource manager. Solution 2 can therefore be applied to all these new transactions.

The list of candidate solutions can be further expended. The point here is that, there are possible solutions, applicability of which is very much depended on the current run-time situation and/or the semantics of the transactions. Obviously, no canonical transaction processing system would exit that serves as the glue of distributed transactions and at the same time provides all the features of the inherently different systems.

A further observation is that some properties like autonomy can be dynamic and dependent on some other features like security. As we explained in the previous section, externalizing the `prepare-to-commit()` operation implies giving the right to the external coordinator to control the used of some local resources (such as data being

locked). Mechanisms are needed to detect and resolve such interactions among different features during run-time.

Based on the above observations, I believe a more flexible and dynamic approach is needed to solve the hard problem. The following hypothetic scenario shows the applicability of reflection to the closed resource manager issues.

```
When a resource manager is invoked by a global
transaction, do the following

if (prepare_to_commit() is externalized)
then {
     // open resource manager
     register to the global transaction;
     execute the invocation}
elseif (the set participating_closed_RMs is empty)
then {
     // apply solution 2
     add the current resource manager to
               participating_closed_RMs;
     register to the global transaction;
     execute the invocation}
elseif (the transaction is read-only) and
        (lower isolation level is allowed)
then {
     // apply solution 3
     register to the global transaction;
     execute the invocation}
elseif (a compensation transaction is provided)
then {
     // apply solution 1
     register to the global transaction;
     make ready the compensation transaction;
     add transaction dependencies;
     // such that the compensation transaction can be
     // triggered when the global transaction aborts
     // after the local transaction commits
     execute the invocation}
elseif (an alternative transaction is provided)
then {
     // apply solution 4
     invoke the alternative transaction}
elseif (the global transaction can be split)and
        (participating_closed_RMs of every new transaction
        has 0 or 1 element)
then  {
     // apply solution 5
     split the global transaction;
     invoke new transactions}
else
     throw exception “cannot resolve closed RM”
```

Notice that in the scenario, reflection at both system and application levels are needed.

- System-level: resource manager openness, support for transaction dependency, isolation levels, split transactions.

- Application level: compensation transactions, alternative transactions, isolation levels, split transactions.

When re-configuration is supported as part of reflection, some of the required functionality can even be built on-the-fly. For example, compensation transactions can be built dynamically.

Other usages of reflective transaction processing include:

- Dynamically support for extended transaction models for more complicated applications.
- Support transaction adaptation for resource-sensitive applications (such as multimedia applications) in changing environments (such as mobile computers).

3 Issues on reflective transaction processing

3.1 Reflection concept

So what exactly is reflection? In general terms, reflection is an entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter. Originally with research on reflection, the entity is the program and the primary subject is the task the program works with. Later on, there have been research activities also in reflective operating systems (such as [18]) and reflective middleware (such as [3]). In this paper, the entity in question is the transaction processing system while the primary subject is the support for transactional properties of transaction programs.

In supporting reflection of a program, two levels of abstraction are needed, namely, the base level and the meta level. If the base-level abstraction is a program itself, the meta-level abstraction then consists of the internal constructs of the program, like types, instances, inheritance, memory in use, ways of initiation, activation, scheduling of threads, or even the current symbol table, runtime stack etc. The two levels are causally connected. Obviously, the meta-level abstraction may again be associated with a meta-meta-level abstraction and eventually these build up an infinite tower of meta-level abstraction.

Reflection actually involves inspection and modification of the meta-level abstraction. The causal connection between the base-level and meta-level abstraction determines how reflective the program can be.

The work on metaobject protocol for CLOS ([10]) is perhaps the most representative on meta-level abstractions. The objective is to supply the programmer of CLOS with a *region* of possible designs – rather than with a fixed, single point – in the overall space of all language designs. In their work, the basic elements of the programming language – classes, methods and generic functions – are made accessible as objects called *metaobjects*. Individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects – a *metaobject protocol*. For each kind of metaobject, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol. In this way, the metaobject protocols, by supplementing the base language, provide control over the language's behavior, and therefore provide the programmer with the ability to select any point within the region of languages around the one specified by the default classes.

What should be included in the meta-level abstraction depends on what is supposed to be observed and eventually modified. Notice that reflection, or modification of the program itself, is only one usage of the meta-level abstraction. Meta-level abstractions that have found wide applications other than reflection include debugging and performance optimization.

Relevant to the issues on heterogeneous transaction processing, it is interesting to notice that meta-level abstractions have already been used to achieve interoperability. In fact, compatibility (as a lesser problem of interoperability) was one of the most important driving forces of metaobject protocols with CLOS. During the development of the CLOS standard, there had been already a variety of object-oriented extensions to Lisp, which were committed to large bodies of existing code. Experience with these earlier extensions had shown that various improvements would be desirable, but some of these involved incompatible changes. Yet, although they differed in surface details, they were all based, at a deeper level, on the same fundamental approach. In another project ([11]), an approach based on a meta object model was proposed to support flexibly object-model interoperability. Unfortunately, to my knowledge, this idea is not further explored. Lately, Microsoft announced the .Net platform [13]. An important feature of .Net is language interoperability. This is achieved by compiling code in various programming languages into the Microsoft Intermediate Language together with plenty of meta data associated with the generated code. Though full reflection is not provided, the meta model, which is essential for supporting reflection, is applied to achieve language interoperability.

3.2 Meta transaction models

To provide reflective transaction processing, a meta transaction model is needed to make explicit the internal constructs of the transaction processing system and provide ways of operating on them. A meta transaction model should define more generic primitives that can actually be used to describe or implement multiple transaction models.

ACTA [4] is a formal framework consisting of primitives for specification of different transactional behavior, namely inter-transaction dependencies (explicit due to transaction structures or implicit due to conflicting operations), manipulation of transactions' views (what the transactions read) and visibility (when and to whom the transactions' effects are visible to other transactions) through the use of *delegation*. Delegation is a very useful primitive in the meta transaction model. It allows a transaction to delegate to another transaction the responsibility of committing or aborting certain operations. For example, committing a sub-transaction can be modeled as delegating all its operations to its parent transaction. When a transaction delegates operations on some objects to another transaction, the former transaction's effects are visible to the latter as part of its view. Views and visibility will also be affected by execution of conflicting operations.

In a rule-based meta transaction model (a description can be found in [9]), transactional behavior is specified by state transitions. Primitives include set of transactions in states like active, aborted, committed, "verbs" for state transitions like begin, commit, rollback, signals for triggering the transitions, and constraints or rules specifying when transitions are enabled.

The ACTA meta model can specify in a finer granularity different transactional behavior whilst the rule-based meta model is more restricted to transaction management

and to some extent recovery. Although the ACTA meta model is more powerful in describing for instance different concurrency control rules, the description is very formal (based on which histories are legal as more relaxed correctness criteria) and thus can hardly be used directly for implementing various concurrency control mechanisms.

ASSET [2] is a more operational form of ACTA. It provides a set of new primitives that enable the realization of extended transaction models: `form-dependency()`, to establish structure-related inter-transaction dependencies, `permit()`, to allow for data sharing without forming inter-transaction dependencies, and `delegate()`, as described earlier. Delegation actually requires "rewriting history". A "lazy" algorithm is implemented by logically rewriting the log on ARIES [12]. However, delegation should include much more than just rewriting the log. Very low-level support in different modules is required (virtually all of resource manager, lock manager, log manager and even buffer or cache managers).

[1] described an implementation of a meta transaction model (a simplification of the ACTA meta model) based on an existing transaction processing product. The meta model is then used to implement some extended transaction models. In the implementation, transaction management adapter and lock management adapters are interposed into the system as an additional level of indirection to support flexibility. Particularly, the effect of delegation on locks was implemented. Dynamic change of transactional behavior is not supported.

There is very little work on whether some different transactional behavior may co-exist. [8] studied how local transaction processing subsystems can be adapted to support global correctness criteria and inter-transaction dependencies.

To support reflective transaction processing, a meta transaction model should not only implement various transactional behavior but also allow the behavior to be modified in a principled manner. This means that the generic primitives should be made explicit and accessible, and safe and consistent ways to operate on these primitives.

3.3 Spheres of control

One difficult problem concerning reflection and dynamic adaptation of transactional behavior is that of "spheres of control". Here a sphere can be understood as a scope containing effects of operations that might be revoked. Basically, transactional behavior is often pessimistic in nature. To assure properties such as atomicity and isolation, certain actions must be taken well in advance, like association of a transaction identifier to every operation on data, writing recovery log records or locking on data items before accessing them. Reflection will imply that histories must be re-played within a certain sphere. Or, some "meta histories" must include information needed for all (or a range of) possible transactional behavior in the future. For instance, delegation implies that locks set to the delegating transaction t_1 must now be set to the delegated transaction t_2, and log records for t_1 must be now for t_2. Furthermore, other parts like caching, paging (which is actually closely related to recovery mechanisms), sessions and database connections etc. will also be affected.

3.4 Incompatible transactional behavior

Certain transactional behavior cannot be simply combined together. For example, different commitment protocols require different log records, forced or not forced (i.e., written immediately to stable storage or later in a lazy fashion). It is also shown that

concurrency control mechanisms that enforce different local atomicity properties cannot be combined [16], because they enforce different serializability orders (some in the order of timestamps of beginning of transactions, some in the order of invocation of conflicting operations on objects).

Even worse, modification of transactional behavior depends not only on the semantics of the transaction in question, but also on potentially all the current active transactions.

A deeper issue is to which extend reflection of transactional behavior is allowed and how consistent transactional behavior is guaranteed during reflection.

3.5 Runtime context

Another important issue is concerned with the level of details of the runtime contexts that vary significantly from system to system.

As an example, *transaction identifiers* are possibly the simplest and most indispensable rumtime context of transactions. At the minimum, a transaction identifier is simply a unique identifier. In most transaction processing standards, a transaction identifier contains two pieces of information: a global identifier of the top transaction and an identifier of the local branch (i.e., at the local database server). In transaction processing products, however, a transaction identifier may also contain (piggyback) additional information, mainly for the purpose of optimization, such as a timestamp of transaction creation, the sequence number of the log record for transaction creation, priority, a tag whether the transaction is read-only, isolation levels, etc.

How to keep track of the resource managers is another example of runtime context. Typically, this information is distributed in the associated transaction managers in a tree-like structure. That is, the transaction manager that creates a transaction is the root of the tree; it keeps track of its children that are the transaction managers that this site invokes. The child transaction managers again keep track of their children, and so on. The consequence of this structure on performance is that registering a newly involved site to the structure is cheap (no additional remote messages are necessary), while the execution of commitment protocol takes long time, since the messages must be propagated from the root to the leaves of the tree. As a different approach, all involved resource managers are registered directly to a single transaction manager. Commitment processing now only has two layers: a transaction manager as the coordinator and all resources managers as participants directly communicating with the coordinator. The performance of commitment processing is improved at the cost of more expensive registration: every resource manager must register directly to the remote transaction manager with a number of remote messages.

There are even subtler runtime details, such as association of transactions with resources (threads, database connections, network connections, distributed caches, replication, etc.).

An important issue is, how much details about runtime contexts should be included in the meta transaction model. The trade-off here is a neat model versus degree of reflection.

4 Reflective transaction management and the Arctic Beans project

From the discussion in the previous section, it is clear that reflective transaction processing is hard. In general, it could be more difficult than any of the particular

problem it helps to solve, such as the heterogeneity issues. However, as we described in Section 1, transaction processing actually consists of three parts. It is often the transaction management part that is highly relevant to distributed systems. Most of the time, we can assume that concurrency control and recovery can be handled locally at every single site. In fact the scenarios shown in Section 2 has nothing to do with specific concurrency control and recovery details.

If only transaction management is concerned, many of the difficult issues we mentioned become irrelevant, such as the subtle issues related with incompatible transactional behavior and spheres of control. It seems to me that reflective transaction management is fairly achievable in a relatively short term.

We have now started to investigate reflective transaction management within the Arctic Beans project. The primary aim of Arctic Beans is to provide a more open and flexible enterprise component technology, with intrinsic support for configurability, re-configurability and evolution. Enterprise components are components at server sites running within so called *containers* which provide support for complicated mechanisms like resource management (threads, memory, sessions etc.), security and transactions. The project will focus on the security and transaction services and how to adapt these services according to the application needs and its environment. More specifically, Arctic Beans has the following objectives:

- To design, implement and evaluate an enterprise component technology based on the concept of containers but with the capability to configure a container at installation time, and to monitor and re-configure this container at run-time;
- To develop an associated meta-architecture for containers, featuring support for structural and behavioural reflection, in order to support the necessary level of openness;
- To develop flexible models of security and transactions within the overall architecture with the goal of supporting a range of applications from large scale business applications to mobile applications;
- To develop a well-defined and complete meta-object protocol for enterprise architectures with a view to influencing the next generation of standards and platforms.

More specific to transaction management, we think a container is a right place to support reflection, because it is the container that interacts both with the application and system-level elements.

5 References

[1] Barga, R. and C. Pu, "A Practical and Modular Method to Implement Extended Transaction Models", in *Proceedings of 21st VLDB*, 1995.

[2] Biliris, A., Dar, S., Gehani, N. H., Jagadish, H. V., Ramamritham, K., "ASSET: A System for Supporting Extended Transactions", in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[3] Blair, G., G. Coulson, P. Robin and M. Papathomas, "An Architecture for Next Generation Middleware", in *Proceedings of Middleware '98*, 1998.

[4] Chrysanthis, P. K., and K. Ramamritham, “ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990.

[5] Chrysanthis, P. K., and K. Ramamritham, “Autonomy Requirements in Heterogeneous Distributed Database Systems”, in *Proceedings of the 6th International Conference on Management of Data*, India, 1994.

[6] Elmagarmid, A. (eds.), *Database Transaction Models for Advanced Applications*, Morgan Kaufman Publishers, 1991.

[7] Garcia-Molina, H. and K. Salem, “SAGAS”, in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1987.

[8] Georgakopoulos, G., M. Hornick, and F. Manola, "Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation", *IEEE Transactions on Knowledge and Data Engineering*, 8(4), 1996.

[9] Gray, J. and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufman Publishers, 1993.

[10] Kiczales, G., J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[11] Manola, F. and S. Heiler, “An Approach to Interoperable Object Models”, in M. T. Özsu et al (eds.) *Distributed Object Management*, Morgan Kaufnamm Publishers Inc., 1994.

[12] Martin, C. P., and K. Ramamritham. “Delegation: Efficiently rewriting history”, in *Proceedings of International Conference on Data Engineering*, Birmingham, 1997.

[13] Microsoft Corporation, *Microsoft .NET*. On-line at: http://www.microsoft.com/net/.

[14] Object Management Group, *CORBA Services Specification*, Chapter 10, Transaction Service Specification, December, 1998.

[15] Wegner, P., “Interoperability”, *ACM Computing Surveys*, 28(1), March, 1996.

[16] Weihl, W. E., “Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types”, *ACM Transactions on Programming Languages and Systems*, 11(2), April, 1989.

[17] X/Open Company Ltd., *Distributed Transaction Processing: The XA Specification.* Document number XO/CAE/91/300, 1991.

[18] Yokote, Y., “The Apertos Reflective Operating System: The Concept and its Implementation”, in *Proceedings of the 7th. Conference on Object-Oriented Programming Systems, Languages and Applications*, ACM Press, 1992.