Coordinated Atomic Actions: How to Remain ACID in the Modern World

Alexander Romanovsky

Department of Computing Science, University of Newcastle upon Tyne, NE1 7RU, UK
alexander.romanovsky@ncl.ac.uk

In the last 10-15 years many models have been proposed to extend conventional ACID (atomicity, consistency, isolation and durability) transactions [1, 2]. Some of the main reasons for this are as follows:

- in some applications data have to be locked for a very long period of time until a long-lived transaction releases them after commit or abort (many extensions allow violations of atomicity and leave the responsibility of tracking the smuggled non-committed information with programmers; some go further and provide support for tracking all dependent transactions which have to be aborted should the transaction from which the non-committed data have been prematurely released be aborted)
- ACID transactions often do not provide suitable recovery techniques (apart from the transaction abort) as they are intended only for tolerating hardware faults (node crashes mainly). This is, for example, one of the reasons for introducing extended transactional models in workflow systems, where one often needs compensation, replacement or alternate actions which can deal with not-committed or erroneous results of a previous action.

If one decides to soften some of the ACID properties he/she has to leave the responsibility of dealing with the consequences of this either with system designers or with a special run-time support. Although some extensions offer very sophisticated support for doing this, there is one important aspect in which system designers are losing here: all benefits of dealing with ACID units. There is a lot of evidence to demonstrate that using ACID (and, in particular, atomic) units facilitates all phases of system development. The system structure becomes simpler when units of system design and execution have atomic semantics; this system is easier to design, to verify, to validate and to understand; providing system fault tolerance (and other means for dependability) is tremendously facilitated if information cannot be smuggled across the unit border.

Recently the concept of Coordinated Atomic (CA) actions [3, 4] has been developed to help system designers to deal with the problems mentioned above while still benefiting from unit atomicity. The uniqueness of this concept stems from the following factors:

- incorporating features for dealing with different types of concurrency
- allowing system designers to deal with faults of different types by applying general exception handling mechanism (which relies on a safe exception handling model, a clear separation of internal and external exceptions and allows for multiple action outcomes [5])

The CA action concept is introduced as a unified general approach to structuring complex concurrent activities and supporting error recovery of multiple interacting objects in a distributed object-oriented system. This paradigm provides a conceptual framework for dealing with cooperative and competitive concurrency and for achieving fault tolerance by extending and integrating two complementary concepts — atomic actions [6] and ACID transactions [1, 2]. CA actions have characteristics of both: atomic actions are used to control cooperative concurrency and to implement coordinated error recovery whilst transactions are used to maintain consistency of shared resources in the presence of failures and competitive concurrency. This allows tolerating faults of various types (as well as their combinations using a resolution mechanism [5]) occurring in different components involved in the CA action execution.

Each CA action is designed as a multi-entry unit with roles activated by action participants which cooperate within the action (Figure 1). Logically, the action starts when all roles have been activated and finishes when all of them reach the action end. The action can be completed either when no error has been detected, or after a successful recovery, or when a failure exception has been propagated to the containing action.

If an error is detected inside an action all roles are involved in recovery. External (transactional) objects can be used concurrently by several CA actions in such a way that information cannot be smuggled among them and that any sequence of operations on these objects bracketed by the CA action start and completion has the ACID properties with respect to other sequences (actions). A CA action execution looks like an atomic transaction to the outside world. The state of the CA action is represented by a set of local and external objects; the CA action (either the action support or the application code) deals with these objects to guarantee their state restoration (which is vital primarily for backward error recovery). Participants cooperate (interact and coordinate their executions) using local objects.

The CA action concept allows designers to reduce system complexity by encapsulating several state transitions of multiple components into one atomic unit with a clearly defined interface. Systems are designed recursively using action nesting; the rules of nesting are straightforward: participants of the nested action must participate in the containing one, a participant can be only in one sibling action at a time, the containing action can complete only after all of its nested actions have completed. Fault tolerance features are always associated with such units which confine errors and are viewed as recovery areas. The action can produce an abort outcome (when, for example, a hardware fault has been detected). CA actions allow system designers to employ design diversity to deal with software design faults [7]. When an action has failed to tolerate an error or to deliver the required results, one of the failure exceptions is propagated to the containing action transferring the responsibility for recovery to a higher system level and leaving the objects involved in the action execution in well-defined states to facilitate the subsequent recovery at a higher level.

CA actions are intended for programming concurrent and distributed systems in which components both cooperate and complete. The conventional ACID transactions are confined to the competitive systems only. To overcome this many extensions introduce specific and restricted ways of cooperation (e.g. some forms of transaction coordination) to make it easier for designers to deal with real systems.

CA actions are units of system structure, system design and system execution. They are highly appropriate for developing complex applications because these structuring units are atomic, and because they allow building systems recursively. Programmers do not have to trace the erroneous information should an action fail because actions are always units of recovery confining all errors.

It is clear that letting potentially erroneous information out of an action (this is what many extensions allow) and tracing it, should the action be aborted, is not suitable for exception handling, which should always be associated with some system structuring units playing the role of exception contexts.

From our point of view, it is very important that CA actions incorporate both clients and servers and impose a structure on both: the components which cooperate (action participants) and transactional objects for which participants from different actions compete. Conventional transactional systems do not usually do this because their main concern is the consistency of servers (transactional objects). A typical example is the CORBA transactional service which allows multiple threads to access transactional objects within an ACID transaction. Although it guarantees the ACID properties of these object state accesses it does not provide any features for multiple thread coordination or structuring. This is why a thread may leave transaction before it has been committed, there is no way to inform all threads about transaction abort, transaction nesting is not in any way related to the thread grouping, etc.

Atomicity of units encapsulating parts of the system behaviour is vital (note that it is very different from the atomicity of separate multicasts found in distributed system research). CA actions (and ACID transactions, for that matter) allow atomicity of multiple operations bracketed by begin-commit or begin-abort constructs. In CA actions these operations are intended for two types of concurrency-related activity:

- accesses to transactional data; this is done in such a way that the whole action is atomic for outside activity (including outside competing actions)
- cooperation among action participants using any suitable methods of message exchange and synchronisation: local shared objects, messages, etc.

We believe that CA actions are suitable for programming long-lived applications because they offer a much more powerful and flexible model than the conventional ACID transactions and allow designers to:

- use exception handling to deal with abnormalities inside actions and to continue execution without having to abort the action
- introduce cooperation explicitly if there is a need to coordinate access to the same objects. In this case system designers have the entire power of any concurrent language (e.g. Java, Ada) at their disposal. It is clear that the only practical solution here is to deal with such problems explicitly and to design a coordination of such accesses. The CA actions allow developers to do this in a disciplined way, enjoying all benefits of ACID properties (and without violating any of them)
- use action nesting and explicitly control/structure system execution: split actions into smaller nested ones, execute sibling actions concurrently, split long actions into sequences of shorter actions or design them as sequences of shorter nested actions, etc.

These features make it possible for actions to live longer and still be atomic.

CA actions provide a wide range of recovery techniques suitable for dealing with different types of faults: node crashes, transient faults, software design faults, environmental faults, etc. There is no need, for example, to violate system atomicity and to introduce any type of auxiliary (e.g. compensation) actions to deal with non-committed or erroneous data outside the action. First, an attempt has to be made inside an action to recover, to compensate or to replace the abnormal activity and state; the exception handling mechanism is a perfect drive for doing this. Secondly, if recovery is not possible and the action cannot produce the requested results, it should signal an exceptional outcome leaving objects in a known state; this external exception has to be dealt with by the containing action using its exception handlers. The containing action is in its turn an ACID unit, all participants of which have to be involved in the recovery should any of them detect an error and raise an exception. Within this approach, different recovery techniques can be used while still dealing with atomic actions.

In the CA action paradigm, systems are designed of objects (or, components) and actions. Actions describe relations (cooperation or competition) in which objects are engaged during system execution. CA action design can be used for both top-down and bottom-up system development, with action atomicity guaranteed. In top-down design, actions are refined into sequences of atomic operations some of which later on can be refined into separate nested atomic actions. In bottom-up development, basic atomic units are glued together in a more complex action of a higher level in such a way that its atomicity is provided. Note that in implementing both types of system design, concurrent programming languages (extended with a CA action API) are used to develop component cooperation, access transactional objects, structure systems of actions, sequence nested actions.

A wide range of CA action schemes have been developed for distributed and centralised applications. These schemes are implemented in Java and Ada, use different types of control (distributed or centralised), employ different policies in dealing with transactional objects (including ones which rely on an ACID transactional system) and different fault tolerance techniques. Recently the CA action concept has been intensively applied to designing a series of safety-critical industry-oriented Production Cell case studies (with environmental and transient faults, real-time constraints, etc.) [8-10] and a non-conventional concurrent computational model (the Gamma computation) [11]. We are now designing new case studies in new application areas, including a distributed internet auction system and a railway scheduling system controlling trains in the vicinity of a station.

Acknowledgements. My thanks go to all former and current members of our team in Newcastle led by Brian Randell.

References.

1 Gray, J, Reuter, A *Transaction Processing: Concepts and Techniques*, Kaufman Publishers, San Mateo, California, 1993

2 Lynch, N A, Merrit, M, Weihl, W E, A, F *Atomic Transactions*, Morgan Kaufmann, 1993

3 Xu, J, Randell, B, Romanovsky, A, Rubira, C, Stroud, R, Wu, Z 'Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery', *Proc. 25th Int. Symp. on Fault-Tolerant Computing,* Pasadena, 1995, 499-508

4 Randell, B, Romanovsky, A, Stroud, R, Xu, J, Zorzo, A F 'Coordinated Atomic Actions: from Concept to Implementation'. CS-TR-595, Department of Computing Science, University of Newcastle upon Tyne, 1997

5 Campbell, R H, Randell, B 'Error recovery in asynchronous systems', *IEEE Trans. on Soft. Eng.* SE-12, 8 (1986) 811-826

6 Xu, J, Romanovsky, A, Randell, B 'Exception Handling and Resolution in Distributed Object-Oriented Systems', *Proc. 18th Int. Conf. on Distributed Computing Systems,* The Netherlands 1998, 12-21

7 Xu, J, Randell, B, Romanovsky, A 'A Generic Implementation Approach to Concurrent Fault-Tolerant Software'. CS-TR-692, Department of Computing Science University of Newcastle upon Tyne, 2000

8 Zorzo, F, Romanovsky, A, Xu, J, Randell, B, Stroud, R J, and Welch, I S 'Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study', *Software - Practice and Experience* 29, 7, 1999, 1-21

9 Xu, J, Randell, B, Romanovsky, A, Stroud, R J, Zorzo, A F, Canver, E, and von Henke, F 'Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions', *Proc. 29th Int. Symp. on Fault-Tolerant Computing*, Wisconsin, 1999, 68-75

10 Romanovsky, A, Xu, J, Randell, B 'Exception Handling in Object-Oriented Real-Time Distributed Systems', *Proc. 1st Int. Symp. on Object-oriented Real-time Distributed Computing,* Kyoto, Japan, 1998, 32-42

11 Romanovsky, A, Zorzo, A F 'Coordinated Atomic Actions as a Technique for Implementing Distributed Gamma Computation', *Journal of Systems Architecture* 45, 15, 1999, 1357-1374

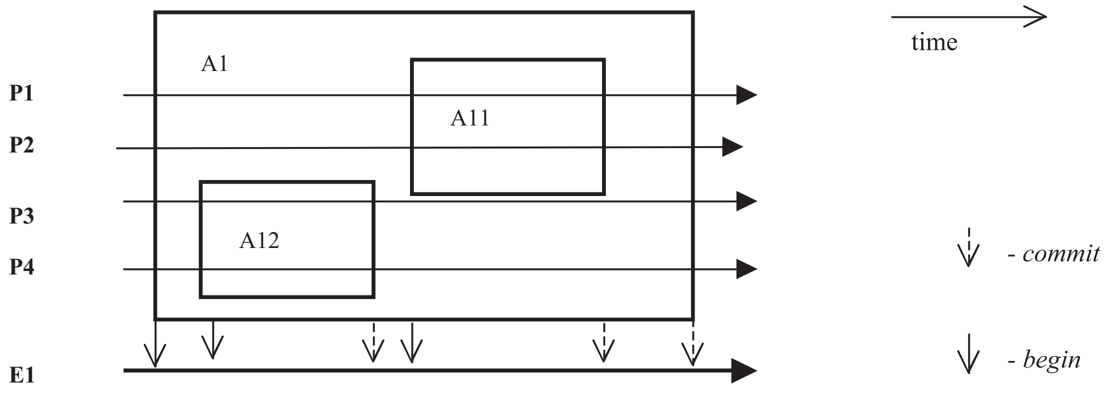

Figure 1. CA actions: participants P1-P4 enter action A1 which has two nested actions A1
operations begin, commit/abort are executed on external object E1