But What if Things go Wrong?

Johan Fabry*
Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussel, Belgium
Johan.Fabry@vub.ac.be

April 5, 2004

Abstract

Building large-scale distributed systems these days is facilitated by a number of distribution frameworks. However, in practice, we see that failure handling is often treated poorly in these frameworks. An exception is transactions, which are a well-known approach for handling failure, and we see that, indeed, these distribution frameworks provide for transaction management. Transactions themselves have a number of drawbacks, which have been addressed by the creation of a number of advanced transaction mechanisms. But, we see that such advanced transaction mechanisms are not used in practice. We think that this is because developing with these mechanisms are not well supported. Therefore, we state that an essential part of a communication abstraction for distributed systems should be a cleanly abstracted way in which to handle advanced transactions, and support for the programmer to use these mechanisms.

Keywords: reliability, advanced transactions, tool support

1 Introduction

Nowadays, building distributed systems is said to be easy: just use one of the many distribution frameworks out there, such as, for example, an implementation of J2EE[12] or CORBA[1], and all the hard stuff will be taken care of for you. Indeed, such frameworks will simplify such things as performing remote method invocations, enforcing security restrictions on method calls, persisting data, looking up services, and so on ...

So, what a programmer needs to learn to be able to implement our Distributed Airspace System, is how to use one such framework by, for example, perusing an EJB book[9]. Once versed in how to write these Beans, and to configure the provided services, which, by the way, in itself, is not a trivial task, all that remains is to write the system in standard OO style. The communication abstractions provided by the framework will take care of all the hard stuff, and we will be home free. Or so the promotional literature will lead us to believe.

*Author funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) in the context of the CoDAMoS project.

But, sadly, life is more complicated than this. In real life things go wrong: the airport network goes down, application servers act up, planes fly out of reach of standard radio communication, et cetera. If we want our Airspace System to be useful, we need to be able to handle partial failures. So lets us examine what the application framework documentation tells us. It turns out that this usually amounts to very little[1], what we find is that we will be notified that something has gone wrong by a thrown exception. However, example code in the literature will usually either ignore this exception, or stop the program, which is clearly inadequate. Also, when browsing through research papers on distributed systems, we all-too often find the equivalent of "the experiment was run on a local network", in other words, the possibility of partial failure was effectively ruled out.

It's easy to see why this is the case: handling failures is difficult. How to handle a failure depends on lots of variables and is usually very application-specific. But, whereas the distributed systems community has progressed considerably, new tools and architectures that address specific kinds of applications being released frequently, the handling of partial failures seems to be stuck in the stone age. In the remainder of this paper we propose one way to evolve one kind of failure handling into something more contemporary. The kind of failure handling we wish to evolve is transaction management.

2 Transaction Management

In fact, transaction management has been with us almost since the stone age: the oldest yet discovered instance of writing is the record of a number of transactions[6]. The concept of transaction is widely disseminated, and again if we look at architectures for distributed systems, we see that many include support for transaction management.

Transactions are a good thing, because they ensure that concurrent processes do not inadvertently work on each others' intermediate results, and generally prevent the underlying data of a system to become inconsistent. This effectively prevents a whole slew of possible errors to occur, which is why transaction management has become quite essential in large multi-tiered distributed systems, of which our Distributed Airspace System would most probably be an instance.

However, transactions are not a silver bullet: while we can prevent inconsistencies in the database, we can't handle the problem of a crashed database server by enclosing database accesses within a transaction. Furthermore, using transactions in itself leads to an extra type of possible failures: transactions may be rolled back by the transaction manager to break deadlocks.

But let us remain focused and simply concentrate on one small aspect of using transactions: how to handle rollbacks in case of a deadlock. Again, looking at the literature for the distributed systems frameworks, we see no thorough treatment of this kind of failure. This could well be because these kinds of failures have been estimated to occur seldom, but if we want a reliable Distributed Airspace System we need to handle them.

Consider, for example an aircraft transmitting turbulence data, to be stored in a database for later retrieval. It makes sense to enclose this in a transaction,

[1] This is not a criticism on the specific book cited above, this problem recurs in the majority of the related literature

to prevent interference with data from other aircraft in the same area. However, it might easily be possible that, due to the turbulence, the radio link is lost for a significant amount of time, and the transaction will take a long time to complete. This has as a well-known issue in that the likeliness of deadlocks increases as the transaction time increases [6]. Also, we can envision other, similar, scenarios, where people use a PDA with a wireless link to book a plane ticket when on the move, and the PDA suddenly loses the network connection. We should be able to handle these kinds of failures, and preferably this should not happen ad-hoc, in a (sub-)application-specific way, but using a more generic model.

3 Advanced Transaction Management

Classical transactions, as described above, have been developed to treat small units of work, which only access a few data items. As a result, as transaction time grows, and the number of data items accesses becomes larger, the performance of the system will drop significantly[6]. This is due to a number of factors, for example the increased chance of deadlocks associated with longer transactions, and the large discrepancy between complex applications and their data requirements and the simple functionality of transactions.

To increase application performance, and to address additional requirements such as cooperation between transactions, a number of advanced transaction mechanisms (ATMS) have been developed[2]. However, each ATMS usually focuses on one single issue, and no overall system has been developed which treats a large number of the identified drawbacks of classical transactions. An impressive number of alternate ATMS can be found in the literature, and two books have been published about the subject [2, 7].

Even if focusing on the issues of deadlocks, due to long term transactions, we can identify different ATMS which handle this issue, for example Sagas[4] and Altruistic locks[10]. In the remainder of this section we briefly outline both mechanisms.

Sagas relax the atomicity requirement of long-term transactions by splitting them into a sequence of atomic sub-transactions. The sequence of sub-transactions should either be executed completely or not at all. Splitting the long-term transactions releases locks earlier, which increases concurrency as other transactions can execute concurrently, and decreases the probability of deadlocks, since after each sub-transaction all locks are released, and each sub-transaction will probably require less resources than the complete Saga.

But what if we want to be able to handle roll-backs of the Saga? In these cases, compensating actions must be executed to undo the effects of already committed sub-transactions. To allow this, the transaction programmer defines a compensating transaction for each sub-transaction. This transaction then performs a semantical compensation action. So to roll back a Saga, the transaction manager aborts the currently running sub-transaction, and subsequently runs all required compensating transactions in reverse order.

An alternative to Sagas is the use of altruistic locks, which is an extension of the two-phase locking protocol that is most commonly used in transaction

[2]Of these, the most well-known ATMS is nested transactions [6]. Briefly put, nested transactions allow for hierarchically structured transactions, and includes rules for nesting the scope of commitment and recovery.

management. When using altruistic locking, a long-term transaction that determines it no longer needs access to certain database objects can release the locks on these objects early, and donate them to other, waiting transactions. This in contrast to two-phase locking, where the first phase of a transaction consists of acquiring locks, and the second phase releases those locks; once a lock has been released, no more locks may be acquired. Using altruistic locking however, a transaction may donate any of its locks, both in the acquisition and in the release phase of the transaction. The donation of a lock signifies that the transaction no longer needs access to that object. This implies that other transactions may access this object concurrently. There are a number of restrictions posed on acquiring donated locks, which must be obeyed at runtime, but we will not discuss them here, as it is outside the scope of this paper.

So, it appears that by using these advanced transaction mechanisms, we have a systematic way in which we can tackle transactional problems, and it appears that we have been talking about a non-issue. But appearances can be deceiving, and it turns out that we do have an issue here.

4 Advanced Transaction Management in the 21st Century

The sad state of failure handling in general also applies to these advanced transaction mechanisms: while these mechanisms have been developed in the 80s and 90s, we still see no use of them in commercial systems. Indeed, we are even hard-pressed to find the most well-known model: nested transactions, in a commercial system.

It would seem that this is the case because implementing a transaction processing monitor which enforces these mechanisms is an unsurmountable task, but this is not the case. Several TP monitors for these ATMS' have been implemented, both in the 80s and 90s [2, 7] and recently [8]. So the problem should lie elsewhere, and given the track record of how error-handling is treated in these kinds of systems, we think we have found the root cause of the absence of the usage of ATMS.

It is our position that the problem with advanced transaction mechanisms lies in the difficulty for the application programmer to specify how to use these mechanisms. Given their nature, an ATMS needs more information about the transaction than a classical system. For example, in Sagas we need to split the transaction in atomic steps and specify compensating transactions, and in Altruistic locking, we need to specify when locks are donated. However, we think that, since letting the developer specify error-handling code is already an issue with current systems, having him go the extra mile to use a more generic ATMS, instead of using an application-specific hack, will be nigh-on impossible.

Therefore, to be able to effectively use these ATMS, we should support the application programmer when specifying these advanced transactions. We envision such tool support by, for example, using a plug-in for an Integrated Development Environment. The plug-in would use some form of reasoning about the code (i.e. code-mining) to determine interesting transactional properties of the code being investigated. The programmer is presented with this information at a higher level of abstraction, and can specify the required transactional prop-

erties at this higher abstraction level, for example by using a Domain Specific Language [13].

We have implemented a first, small, prototype tool[3] for use in Enterprise JavaBeans[11] as a first step toward this goal. Our tool can detect methods that should be made transactional, based on the types of beans being accessed and if getters or setters are called on these beans. Also, again by using code mining, our tool can suggest possible compensating methods for the current method, allowing the programmer to pick one from a list. Furthermore, our tool includes a simple extra that facilitates the handling of deadlocks: the programmer can specify that in such cases the transaction should simply be restarted, by restarting the corresponding method.

We are currently extending our tool, and considering the following avenues for further work: First we are looking at what more useful information can be gathered from the source code, which we can bring to a higher abstraction level and present to the programmer. Second, we are considering what other generic deadlock- or exception-handling strategies could be worthwhile to offer the programmer. Third, we are perusing the work on coordination languages[5] to investigate how transaction sequences can be specified, for use, in for example, Sagas.

5 Conclusion

In this paper, we have talked about failure management in a large scale distributed system, such as, for example, a Distributed Airspace System, and how the tackling of this aspect seems to be somewhat lacking in current day systems. We have focused on transaction management as a widely-used means of performing a subset of failure prevention and failure management. We identified a weak point of transaction management, namely the problem of deadlocks, especially in long-term transactions.

We have shown that a number of solutions exist for the drawbacks of transaction management, in the form of advanced transaction mechanisms. Many such mechanisms have been developed, and we briefly touched on two of them: Sagas and Altruistic locks. However, while this research was performed a while ago, we see that these models have not been put into practice.

Our position statement is that this is because such advanced mechanisms are too difficult to use by the application programmer without some form of help: they require too much advanced information to be specified. Therefore, we have implemented a prototype tool to help the programmer, by allowing him to reason about such forms of transaction management at a higher level of abstraction. This is achieved by mining transactional information hidden within the source code, and allowing specification of properties at this high level of abstraction through a domain-specific language.

6 Acknowledgments

Thanks to Thomas Cleenewerck for his helpful comments and Theo DHondt for supporting this research.

References

[1] The common object request broker: Architecture and specification. Technical report, Object Management Group, Inc., 2002.

[2] A. K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.

[3] J. Fabry. Transaction management in EJBs: Better separation of concerns with AOP. In *The Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, 2004.

[4] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[5] D. Gelernter and N. Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[6] J. Gray and A. Reuter. *Transaction Processing, Concepts and Techniques*. Morgan Kaufmann, 1993.

[7] S. Jajodia and L. Kershberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[8] R. Karlsen. An adaptive transactional system - framework and service synchronization. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*. Springer Verlag, 2003.

[9] R. Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, third edition, 2001.

[10] K. Salem and H. Garcia-Molina. Altruistic locking. *ACM Transactions on Database Systems*, 19(1):117 – 165, March 1994.

[11] Sun Microsystems. Enterprise JavaBeans 1.0 architecture. http://java.sun.com/products/ejb/docs.html.

[12] Sun Microsystems. Java 2 platform, enterprise edition. http://java.sun.com/j2ee/.

[13] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.