

Inconsistency Tolerance across Enterprise Solutions

Peter Henderson, Robert John Walters and Stephen Crouch
Declarative Systems and Software Engineering Group
Department of Electronics and Computer Science
University of Southampton
Southampton,
UK.
SO17 1BJ

Abstract

As every information system becomes connected to every other information system, they form the so-called "information utility". This is the domain in which contemporary distributed systems have to operate. New applications have to be evolved on this platform of existing systems that may hold inconsistent information. Consequently, solutions need to be able work in a world of only partially correct information.

In this paper, we discuss means whereby architects, designers and engineers may, in this context of information inconsistency, develop new business solutions and reason about their validity. In particular we describe the properties of inter-enterprise system architectures for applications working with partially replicated and partially consistent information. These must be able to operate under reversible assumptions and to undo operations as a consequence of reversing assumptions. We have developed exemplary architectures that exhibit these properties, used them to investigate the concept of inconsistency-tolerant components and begun to devise methods of building inter-enterprise applications from such components.

This approach, we conjecture, makes reasoning about the validity of proposed inter-enterprise scale solutions more straightforward and thus increases the speed with which new solutions can be deployed. We are evaluating these ideas now, by building, along with our industrial collaborators, realistic enterprise-scale demonstrations in the domains of Finance and Defence.

1: Background

The growth of the everything-connected-to-everything-else information utility arises now because of the ubiquity of distributed computing, in particular the internet, both within an organisation and between organisations [14]. Indeed, the fact that the global internet and intranets of individual companies are based on the same technologies, has made business-to-business e-commerce a growing reality [4]. The problem this creates is that we have to evolve business systems from existing components [12], where the information that they integrate comes from multiple sources and this information is often inconsistent across these sources. The sheer scale of the systems that must be integrated in order to achieve business objectives exacerbates the problem [1]. It is on this scale that we need to be able to reason about the validity of solutions.

Overall, we need to maintain known consistency properties of partially replicated information in order to support continuous potential-for-change and evolution. This is not an uncommon circumstance in Transaction Processing (TP) [3] systems, but TP systems are typically very tightly coupled and consequently do not allow easy evolution. They implement strong consistency using protocols for distributed TP such as two-phase-commit. We need to challenge the appropriateness of two-phase-commit and other quasi-synchronous transaction models, partly because of the consequences for reliability and performance, but in particular in the context of the information utility, because of the reality that data sources will frequently be inconsistent with each other.

Hence the move to more loosely-coupled application systems integration, with technologies based on relatively loosely-coupled technologies derived from XML-messaging. Such approaches introduce asynchronous communication with concurrency, which adds a difficult temporal dimension into reasoning about information consistency [6, 11]. Enterprise Application Integration (EAI) tools provide *mechanism(s)* for integration but there is little formal underpinning. Reasoning about validity of a proposed solution is largely left to the intuition of the designer. In order to improve this situation, we need to be able to:

- Categorise component application system properties in terms of information storage & access
- Reason about adding new components to existing configurations
- Reason about migration of information (and functions)
- Reason about consistent derivations of information
- Reason about the consistency properties visible to an observer with access to several components

These requirements lead us to consider the properties that an enterprise system must have if it is to be very flexible (allowing, for example, enterprise-level plug-and-play) [7, 12]. Before we enumerate those properties and discuss architectures which support them, we will look at a simple example from the domain which interests us.

2: An Example

To be more specific about the nature of information consistency, consider the example shown in Figure 1. This shows the sequence of messages that might flow between an Electricity Consumer, their Electricity Supplier and two Banks used by the supplier and the customer respectively. The example, in the form of a UML collaboration diagram, shows the messages used to complete the payment of a quarterly bill. The messages themselves may take a few minutes, but the total transaction time from beginning to end will probably be a few days. Even when operating properly, there are inconsistencies between the world-views of each participant until the whole extended transaction is complete.

When messages get lost or arrive in unexpected sequence, or incorrect information is sent, these inconsistencies will persist for a long time, and may require human-intervention to be corrected. The whole problem is exacerbated by the fact that each participant is

involved in many such extended transactions with others (not shown in Figure 1). Consider what would happen if, before one of the above messages had arrived, one of the participants acted on their assumption that it had been received. They would be acting on an assumption that their view of the world was consistent.

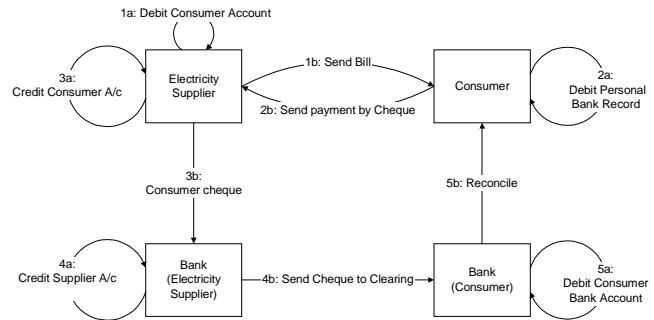


Figure 1: Collaboration Diagram showing Extended Transaction

When a business, supplying services into an environment as complex as this, decides to launch a new service, they need to devote substantial effort to ensuring that *information consistency* is eventually achieved. In situations such as this, conventional transaction processing (vital though it is near the databases of each participant) doesn't have the answer. We need new concepts of what it means for information to be consistent, how to know when that has been achieved and how to devise systems that are *inconsistency-tolerant*. That will greatly reduce the cost of inter-enterprise business evolution and speed up the rate at which businesses themselves can evolve.

Let us look briefly at the world from the point-of-view of the Electricity Supplier. They have some data, held locally, about the Consumer's account with them. On issuing a bill (1b) they debit their record of the Consumer's current balance (1a). Later, the payment arrives from the consumer in the form of a cheque (2b). The Electricity Supplier then makes a credit to its record of the Consumer's account (3a) and forwards the cheque for processing by its own bank (3b). That would be a relatively normal sequence of events.

All sorts of problems could arise, of course. Some of these would be the consequence of faults and would be remedied by the normal structures for fault-tolerance. We are more interested in the problems that arise because information is not consistent. Generally, there are two types of information inconsistency, which we can refer to as logical inconsistency and temporal inconsistency. In terms of the example we have just given a logical inconsistency would arise if, for example, the payment referred to an account that did not exist. A temporal

inconsistency would arise if, for example, a payment arrived before a bill had been sent. A component may or may not be tolerant of these forms of inconsistency. And, of course, the inconsistency which arises between components is also important. For example, the Consumer's view of their balance at the Electricity Supplier and the Electricity Supplier's view will not always be the same. Tolerating this form of inconsistency, especially if it is a long time before it is resolved, is also of importance to us.

3: Properties of Architectures for Inconsistency Tolerance

Let us now look at some of the *properties* that must be supported by an inconsistency-tolerant, inter-enterprise architecture. For example, such an architecture must have the ability to operate under reversible assumptions and the ability to undo operations as a consequence of reversing some assumptions.

In terms of our example, suppose the Electricity Supplier receives a payment that is not expected and the amount of the payment is not equal to the amount outstanding. The Electricity Supplier's response could be just to credit that to the Consumer's account anyway. On the other hand, the Electricity Supplier could do more. For example, it could notify the Consumer of this inconsistency. It probably would do so if the inconsistency was an underpayment. It might make good business sense to also do so when the inconsistency was a significant overpayment, because the overpayment suggests an inconsistency at the Consumer's end.

This example suggests some significant properties that an inconsistency-tolerant component should possess:

1. A component should be prepared to accept messages (instructions) in any order whatsoever. It should not, for example, only be prepared to accept message A as a response to having sent message B. In terms of our example, payments should be accepted by the Electricity Supplier at any time and for any amount.
2. A component should be able to operate on the basis of making *assumptions* about the rest of the world, where the information it has about others is inconsistent. In terms of our example, the Electricity Supplier could assume that an overpayment was intentionally high, or that it was unintentionally high and is allowed to behave accordingly.
3. Having operated on the basis of an assumption, a component should always be in a position to undo the effects of any actions it has performed on the basis of that assumption. For example, the Electricity

Supplier, having assumed that an overpayment was intentional should be able to reverse that assumption when receiving a later request from the Consumer for a refund to correct their mistake.

4. Recognising that some actions which are performed on the basis of an assumption are not going to be entirely reversible, the component should have a scheme whereby it can eventually reach a consistent state even though it never would have reached the present state had it made a different assumption. We will give a detailed example of this in a moment.

One consequence of property 1, that components should always be able to accept messages of any type and at any time, is that any architecture that supports this type of behaviour will be equivalent to what we call an *inbox* architecture. That is, a user of the component can deposit messages and not have to wait for a reply. The archetypical implementation of this architecture is email, but any message queuing technology (e.g. MQ, JMS, or even ad-hoc XML messaging over HTTP) can be used to implement an inbox architecture. This is distinct from a remote object architecture (RMI architectures such as Corba [10, 15] etc) in that there is no acknowledgement to the user and no reason for the user to wait. The user is not blocked. Clearly one can build an inbox architecture out of RMI quite trivially, just make sure every method returns immediately having noted the request, but the notion of inbox is conceptually different from the notion of RMI.

Consider now the problem that arises as a consequence of property 4, where an irreversible action is performed. Figure 2 shows a state chart of a component which is performing actions *a*, *b*, etc. We denote by \bar{a} the action which undoes the effect of *a*. Starting in state 1, we assume the component performing these actions has made an assumption that leads it to perform action *a* in preference to action *b*. This is OK, because if at state 2 it discovers that the assumption was wrong it can undo *a* and then do *b*.

Even if it continues via *c* to state 3, it can reverse the assumption made at state 1 by first undoing *c*, then undoing *a* and finally doing *b*. But suppose that the dotted line in Figure 2 is a line that, once crossed cannot be re-crossed. Having performed action *a* in state 1, to reach state 2, discovering that this was a mistake is not reversible. The only course of action is to head for a state which could have been reached from state 1 had the right action (*b*) been performed. On figure 2, this is to perform *c e* from state 2 to reach state 5 which is a consistent place we might have reached from 1 had we done *b d*.

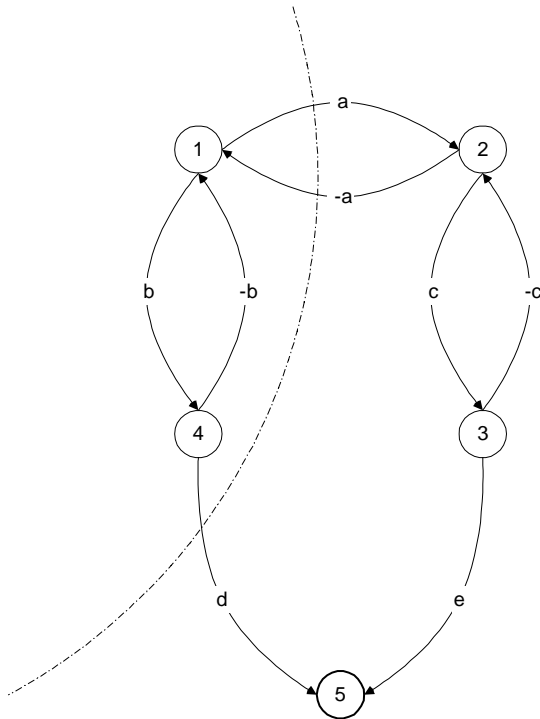


Figure 2: State Diagram showing reversible actions

This kind of reasoning is very familiar from the models that we have in distributed systems [8, 9] for reaching *consistent global states*. These models introduce increasingly sophisticated logical clocks in order to present the means whereby consistent values of distributed objects can be obtained for evaluation. There are problems (e.g. distributed deadlock detection) where such consistent views of distributed state are essential. Our view is that in enterprise systems this is often a luxury we can not afford and that we have to support computation which tolerates inconsistency over potentially extended periods of time. In the next section we discuss an architectural pattern which we are developing to support distributed computation with components which are tolerant of inconsistent information. It is a version of, although not the only possible version of, an inbox architecture.

4: An Architecture for Inconsistency Tolerance

Consider the system component shown in Figure 3. This shows a (legacy) business application B which has been made into an EAI component by the addition of a messaging interface M. Since it must communicate with other components of the same structure, it publishes the

message formats which it will accept (coming in via M) using something such as XML Schemas. It also takes responsibility for transforming its outgoing messages into the form required by the component which it is talking to. This is the role of A in Figure 3.

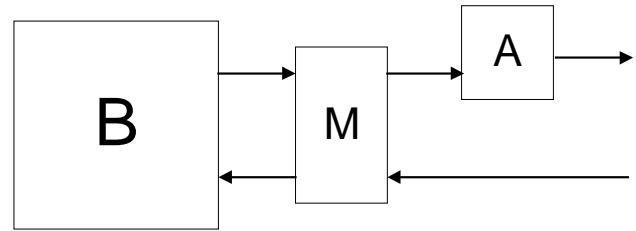


Figure 3: A Transactional System with Messaging Interface

You can see how such a component could work with others to implement inter-enterprise business processes. Systems of this sort are being built using contemporary EAI mechanisms such as BizTalk, UDDI and WSDL[2, 5, 13]. But there is a very strong assumption being made about consistency. The messages sent through A, to be received by someone else's M, have to be *logically consistent* (cf. type correct) and *temporally consistent* (e.g. arrive in an expected sequence). Getting systems to work with this strong consistency is very expensive and very error-prone.

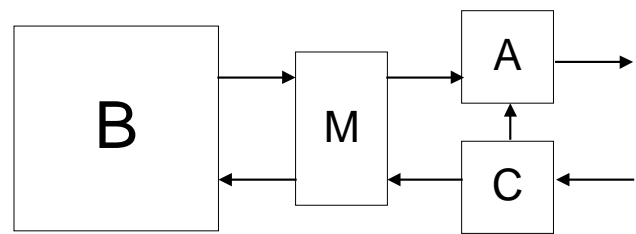


Figure 4: Made Inconsistency-Tolerant

What we need is for our EAI components to be tolerant of inconsistencies with respect to message content and order of arrival. One way in which this could be achieved is to add a filter C to the incoming message stream (see Figure 4), with the property that it is tolerant of messages that are inconsistent with respect to the expectations of M. This filter can be as constructive as we like. It can re-sequence messages, or negotiate over missing, incorrect or mismatched data. Whatever it does, we can describe its behaviour in terms of its ability to make *assumptions* that it will, if necessary, be able to reverse. For example, if an order arrives from an

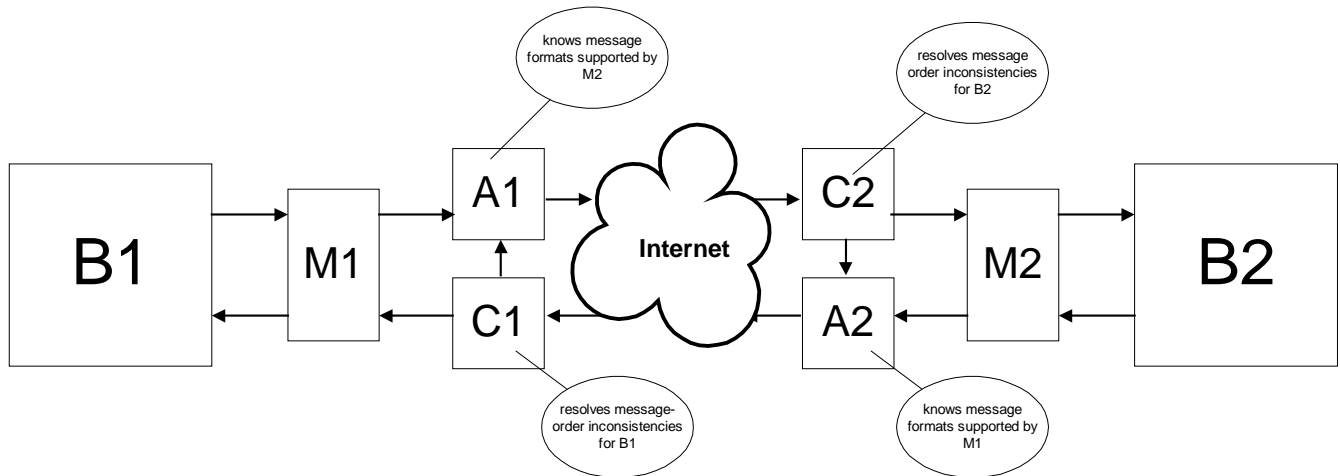


Figure 5: What each System assumes of the other

unregistered customer, but M expects customers to be registered, C may assume that the customer intended to register and make the registration on their behalf. If it subsequently transpires that the customer has in fact sent the order to the wrong supplier, C will be able to unpick the consequences of the transactions that it has carried out on B.

Figure 5 shows the assumptions that each of the components makes about the others. Reasoning about the validity of a solution with this architecture is simplified by the fact that the original applications (B1 and B2) will only see the real-world through the interfaces provided by C1 and C2. Reasoning about the interaction between C1 and C2 is of course where the complexity lies. Here we must ensure that progress is made in the extended transaction, while protecting B1 and B2 from the abuse of inconsistent information.

5: Conclusion

With the approach of the "information utility" in which every information system will be connected to every other, data is becoming distributed and replicated. However, many of the connections between systems are either loosely coupled or unreliable. As a consequence, we cannot enforce consistency using transaction processing-like schemes. Instead, we have to accept that inconsistencies in data are unavoidable and contemporary distributed systems will have to be able to cope in such an environment.

When these systems encounter problematic data, they need to be able to proceed despite the shortcoming. To do this, they have to make assumptions but, where

they do, they must also be able to handle the situation when an assumption needs to be reversed.

In this paper, we have discussed some means whereby architects, designers and engineers may develop new business solutions and reason about their validity. We have also described some properties that an inconsistency-tolerant component should possess.

We have begun to devise methods of building inter-enterprise applications from such components with these properties and believe this approach makes reasoning about the validity of proposed inter-enterprise solutions more straightforward.

References

- [1] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool, "Replication, consistency, and practicality: are these mutually exclusive?," *SIGMOD Record*, vol. 27, pp. 484-95, 1998.
- [2] Ariba Inc, International Business Machines Corporation, and Microsoft Corporation, "Universal Description, Discovery and Integration - Technical White Paper," 2000. See: www.uddi.org.
- [3] P. A. Bernstein and N. Goodman, "An algorithm for concurrency control and recovery in replicated distributed databases," *ACM Transactions on Database Systems*, vol. 9, 1984.

- [4] L. Cardelli, "Abstractions for Mobile Computation," in *Secure Internet programming*, vol. 1603, J. Vitek and C. D. Jensen, Eds. Berlin: Springer-Verlag, 1999, pp. 51-94.
- [5] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web Services Description Language (WSDL) 1.0," 2000. *See:* <http://www.ibm.com/developerworks/library/w-wsdl.html>.
- [6] A. Dickman, *Designing Applications with MSMQ - Message Queuing for Developers*: Addison Wesley, 1998.
- [7] P. Henderson, "Laws for Dynamic Systems," presented at International Conference on Software Re-Use (ICSR 98), Victoria, Canada, 1998.
- [8] P. Henderson and R. J. Walters, "Behavioral Analysis of Component-Based Systems," *Information and Software Technology*, vol. 43, pp. 161-169, 2001.
- [9] P. Henderson and R. J. Walters, "Component Based systems as an Aid to Design Validation," presented at 14th IEEE International Conference on Automated Software Engineering (ASE99), Cocoa Beach, Florida, 1999.
- [10] JavaSoft, "Java RMI specification," 1996. *See:* <http://www.javasoft.com>.
- [11] B. Kent, "The Semantics of Object Identity," 1992. *See:* <http://home.earthlink.net/~billkent/catalogsource.htm>.
- [12] F. Leymann and D. Roller, "Workflow-based applications," *IBM Systems Journal*, vol. 36, 1997.
- [13] Microsoft Corporation, "BizTalk.Server Homepage," 2001. *See:* <http://www.microsoft.com/biztalk/default.asp>.
- [14] L. Nicolle, "John Taylor - The Bulletin Interview," *The Computer Bulletin*, 1999.
- [15] Object Management Group, "Common Object Request Broker: Architecture Specification," *See:* <http://www.omg.com>.