

Ensuring Dynamic Reconfiguration Consistency

N. De Palma ^{*}, P. Laumay ^{*}, L. Bellissard [§],
INRIA Rhône-Alpes, SIRAC project
655, Avenue de l'Europe
F-38330 MONTBONNOT SAINT-MARTIN, France
Tel: +33 4 76 61 53 64, Fax: +33 4 76 61 52 52
Email: Noel.Depalma@inrialpes.fr

Abstract

The evolution of distributed applications to reflect structural changes or to adapt to specific conditions of the run-time environment is a difficult issue especially if continuous service is required from end-users. This latter constraint implies to perform changes with minimal penalty on the service provisioning. The set of tools and services that allow such a goal to be achieved is usually designated as dynamic reconfiguration capabilities. A major issue related to dynamic reconfiguration is to ensure applications consistency after a reconfiguration. Classical transactional models provide a way to identify application calculations and to ensure consistency of these calculations despite faults. In a first step, we propose an extended transaction model to ensure such a consistency when a reconfiguration occurs. We argue that ensuring consistency can be done easily by using an extended transaction model with a strong isolation property. Then we discuss possible solutions for non transactional applications.

1 Introduction

The growing use of the Internet as an execution environment for distributed applications emphasises the problem of managing the evolution of an application to meet its various users requirements. The logical structure of the application itself may evolve over time to reflect changes in the services provided to the users (e.g. a new service is made available or a new version of a service is replacing an old one). The physical structure of the application may also change to adapt the overall application's behaviour to evolving conditions of the run-time environment (e.g. node or network failure, disconnected mode for a mobile user, ...).

Dynamic reconfiguration refers to the set of tools and services allowing these changes to be performed in a dynamic way (i.e. without stopping the entire

application). The overall objective is to freeze only the part of application concerned by the modification so that the overall penalty on the running application is minimized. Dynamic reconfiguration covers the four following changes:

- Modifying the architecture of an application (adding/removing components, or modifying the interconnection pattern);
- Modifying the geographical distribution of an application (changing the placement of components);
- Modifying the implementation of some components;
- Modifying the interface of some components;

So far we have only considered the first two kinds of modification.

A major issue related to dynamic reconfiguration is to ensure the application remains consistent after a reconfiguration. A distributed application is defined by a set of global distributed calculations. An application is consistent if the results of running calculations are not modified when a reconfiguration occurs.

In order to ensure consistency, we propose a way to capture the structure of application computations and to constrain component reconfiguration according to this structure. Our solution is inspired from distributed transaction models. In these models, transactions are used to identify integrity constraints that must remain consistent after a fault. We propose to extend a classic transaction model to identify the global calculations that must remain consistent after a reconfiguration.

In a first step, our target applications are transactional applications built as a set of components. We argue that ensuring consistency with transactional applications can

^{*} Affiliated to Joseph Fourier University, Grenoble

[§] Affiliated to INPG, Grenoble

be done easily by using an extended transaction model with a strong isolation property. Then we discuss possible solution for non transactional applications.

The paper is organised as follows. Section 2 provides an analysis of reconfiguration related problems. Section 3 describes our dynamic reconfiguration mechanisms. Section 5 is a short discussion about the generalization of our approach. Finally we conclude in section 6.

2 Overview of reconfiguration problems

The challenge is to provide dynamic reconfiguration mechanisms which maintain applications consistency while minimizing the impact on the running applications.

We make a distinction between local consistency and global consistency. Local consistency is relevant to the inner computation of a component whereas global consistency is concerned with global distributed application calculations.

2.1 Local consistency

Three basic problems leading to inconsistencies during reconfiguration can be underlined. These problems are related to the naming of components, their state and communication channels. These problems lead to inconsistency of component local state.

The following example illustrates some situations which can imply inconsistencies. In this example, component A2 moves from node 2 to node 3.

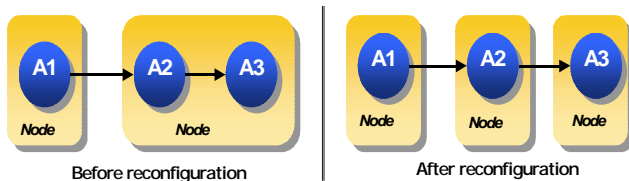


Figure 1 : Migration of A2 from node 2 to node 3

At reconfiguration time, the following issues have to be considered in order to maintain consistency:

- **Naming preservation**

The first issue is related to naming problems: component A1 uses a reference to A2. When component A2 moves from node 2 to node 3, a reference to A2 may become wrong if names include information about location. As a consequence component A1 cannot communicate with A2 anymore. The local state of A1 contains an invalid reference to component A2.

- **State preservation**

Another issue is a state preservation problem. When component A2 moves to node 3, its current state must be preserved. This means that component A2 can finish its actual computation on node 3 from its former state. If reconfiguration affects only application geometry (e.g.

migration) then the computation results are the same as the ones obtained with no reconfiguration. A component which cannot achieve its computation from its previous state is inconsistent. The local state of A2 must be preserved in order to carry on its execution on the new node.

- **Status of communication channels**

The last issue is related with communication channels: When a component moves to another node, some messages may still be in transit and might be lost. In figure 2 we see that when component A2 moves from node 2 to node 3, m3 is "in transit". This message should reach node 2 after its migration to node 3 but is lost since A2 is no longer on node 2.

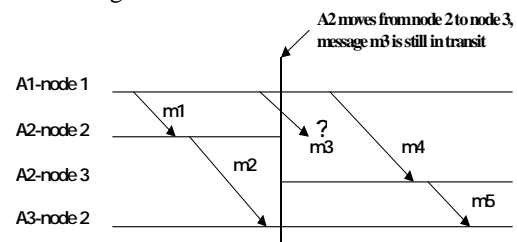


Figure 2 Status of communication channels

In this section we have outlined some basic problems that may entail local inconsistencies during reconfiguration.

Fortunately, many solutions to these problems have been proposed especially in the area of process and object migration. Each of these solutions has its own advantages and drawbacks. Examples of these techniques include :

- Forwarder techniques are used to handle messages in transit during migration.
- Location independent naming mechanisms may solve naming problems.
- Lazy update mechanism for references held by components. Basically, when a component gets an exception during a remote communication, the migration of the target component is suspected and the communication initiator consults a naming server to obtain the latest component reference.
- Checkpointing is a technique which allow component state preservation.
- Java serialization is another technique used to save and restore object state.
- Thread capture mechanism allows a fine grained state capture by pickling thread state.

Our goal is not to develop new mechanisms to ensure local consistency but to apply existing techniques coming from process migration research to a broader reconfiguration topic. Detailing more precisely migration technique is out of the scope of this paper since we focus on solutions ensuring global application consistency during reconfiguration. For more information about these migration techniques, the reader can refer to

[6][7][8]. The following section gives an example of such global consistency related problems.

2.2 Global consistency

Local consistency can be achieved by resolving the basic problems presented above. However, this is not sufficient since consistency can also depend on the applications' global calculations, which must remain consistent despite reconfiguration. The following example (an e-commerce scenario) illustrates such a consistency problem. This application is composed of an e-commerce front-end *Server*, a *Process* component which handles client orders (order processing, billing ...), and a set of databases (*db_session*, *db_client* ...). *Db_session* stores temporary information about client's sessions while *db_client* stores persistent information about them.

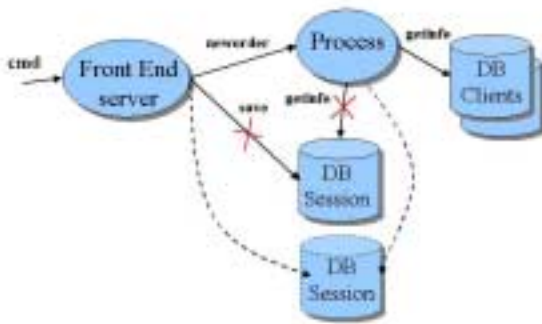


Figure 3 : An e-commerce example

In this example, the only global calculation can be informally expressed like this :

- On clients' order, the server saves information related to client sessions in database *db_session* and then notifies process component about the new order.
- On new order, the *Process* component starts processing the client's order. This may include information extraction from databases *db_session* and *db_client*.

The reconfiguration scenario related to this example is the replacement of database *db_session* by a new version. This reconfiguration scenario involves components *Server*, *Process* and database *db_session*. All connections which point to old *db_session* database must be rebound to the new database version. Since *db_session* only contains temporary informations about clients' sessions, we do not transfer session-related informations to the new database. This means that we cannot replace database *db_session* by a new version whenever we want. We must take care about the global computation to reconfigure it safely. When such a reconfiguration occurs in our scenario, we must ensure that all temporary information previously stored by the *Server* component (*save operation*) have been taken into

account by the *Process* component. If this constraint is not fulfilled, the *Process* component will not retrieve the last client session information since the *Server* component has written these information into this previous database version.

Ensuring global consistency requires a way to capture global calculations and to constraint component reconfiguration according to this calculations. For example, in the previous scenario, we cannot reconfigure component *Process* when the global calculation is running.

3 Ensuring global consistency

In this section we will analyse how to ensure global application consistency when a reconfiguration occurs. As we have seen, local consistency can be achieved by solving the basic problems presented in section 2.1. However, this is not sufficient since consistency can also depend on the applications' global computation.

3.1 Analysing consistency constraint

We propose to capture global calculations by identifying causal communication flows between components. For example in the scenario given figure 3, we can identify the calculation depicted in figure 5:

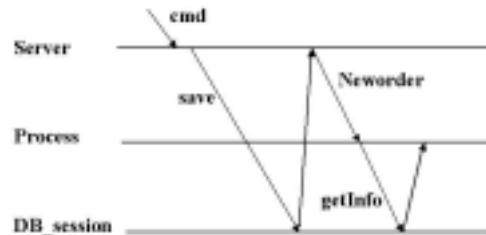


Figure 5: A computation flow

Let us call this calculation "C0". The question remaining is how to identify communication flows and how to control them in order to maintain application consistency when a reconfiguration occurs.

In our mind, identifying communication flows is similar to transaction demarcations in classical transactional systems. As a consequence, we propose to use an extended transaction model to identify global calculations and to ensure application consistency in the face of reconfiguration. Assuming that calculation C0 of our example is now modeled as a transaction, reconfiguration constraints must disallow connection modification between process component and database *db_session* during the progression of this transaction.

3.2 A first step toward an extended transaction model

In this section we present an extended transaction model used to ensure reconfiguration consistency.

In a first step, we will distinguish between two kinds of transactions :

- Reconfiguration transactions which run a set of reconfiguration operations. Let RT be the set of this kind of transactions.
- Application transactions which require ACID property and a special case of isolation with reconfiguration transactions. Let AT be the set of this kind of transactions.

To ensure application consistency during a reconfiguration, we need an extended isolation property between a reconfiguration transaction and other application transactions. This isolation property ensures that reconfiguration transaction will be applied on a consistent global computation.

The two kinds of transactions have the following properties:

- Transactions belonging to AT have the classical ACID properties. Furthermore they are isolated from each other (classical isolation) but they are also strongly isolated (defined below) from transactions belonging to RT .
- Transactions belonging to RT have the classical ACID properties but they are strongly isolated from other reconfiguration transactions and from transactions belonging to AT . Modeling these kinds of transactions is a way to identify global application calculations.

Def 1: Conflict

Let Oa be the set of objects involved by a reconfiguration transaction tr and Ob the set of objects involved by a transaction t ($t \in RT$ or $t \in AT$). Transaction tr is in conflict with transaction t if $Oa \cap Ob \neq \emptyset$.

The required isolation property between reconfiguration transaction and application transaction is the following :

Def2 : A strong isolation property

A reconfiguration transaction Tr must run in an exclusive manner with respect to the set of transactions that are in conflict with Tr .

This isolation property ensure that no application transactions become inconsistent due to a reconfiguration transaction.

One basic way to ensure this property is as follows : when a conflict is detected between a reconfiguration transaction tr and another transaction t , the transaction manager can either rollback the transaction t or the reconfiguration transaction tr . To implement this solution, we need to implement an extended transaction model which provides a strong isolation property between different kinds of transactions. This requires to adapt common transaction manager as follows :

- The transaction manager needs to differentiate reconfiguration transactions from application transactions. For this purpose, we propose to introduce two different BEGIN tags used by programmers to note transactions beginning. We provide the BEGIN_R tag to tag reconfiguration transactions and the BEGIN tag to tag common application transactions.
- The transaction manager must handle the isolation property. This can be achieved by adapting lock manager to detect and resolve conflict between reconfiguration transactions and other transactions.

This can be implemented as a special case of a deadlock prevention algorithm implemented by the transaction manager. In the following we present the initial wait-die algorithm[9] used to prevent deadlock then we give an extension of this algorithm to ensure global consistency toward a reconfiguration.

Initial wait die algorithm

Transactions are ordered with timestamps. A transaction is stamped on its creation. The order between different transactions is identical on all execution nodes. Let Ti and Tj be two transactions respectively marked with stamps ei and ej . We express transaction order as follows :

$$Ti \text{ before } Tj \iff ei < ej$$

When a conflict occurs, the order established between transactions induces either a rollback or a wait for the resource. This method avoids deadlock by forcing transactions to wait for a resource according to their order.

Let Ti and Tj be two transactions respectively marked with stamps ei and ej . Let R be a resource locked by transaction Tj . Since Ti also requires to lock R , there is a conflict between Ti and Tj . The initial wait-die algorithm is the given bellow :

```

If (Ti before Tj) then Ti wait
else Ti is rollbacked
```

Figure 7 : Initial wait-die algorithm

Reconfiguration-aware Wait-Die

In the following we present an extension of the deadlock prevention algorithm enhanced to ensure strong isolation property.

```

if (Ti before Tj)
  If (Tj == reconf) Ti is rolledback
  else if (Ti == reconf) Tj is rolledback
  else Ti wait
else if (Ti = reconf and Tj!=reconf)
  Tj is rolledback
  else Ti is rolledback

```

Figure 6: Reconfiguration-aware wait die algorithm

This algorithm is based on the fact that when a transaction t is conflicting with a reconfiguration transaction tr , t is rolledback despite the order between t and tr . This algorithm ensures that application transactions remain consistent toward a reconfiguration transaction.

Programmers just have to identify global calculations that require to be consistent toward reconfiguration. The resulting programming model is close to the classical transaction programming model since we just have to differentiate between reconfiguration transactions and application transactions. Furthermore, since our target applications are transactional by nature, it is very easy to implement this algorithm. An elegant way to implement this extended transaction model would be to modify an open transaction manager that allows us to change its standard application-level interface and to replace its deadlock algorithm by our Reconfiguration-aware algorithm. The following section deals with this issue.

3.3 Technical issues

A way to implement this extended transaction model would be to use an adaptable transaction manager which provides an open implementation of its inner functionalities such as lock management. Opening up a TP monitor is characterized by two inter-related problems: The interface for customization and the level of customization. Some works have focused on providing transaction manager adaptability by using reflection mechanisms in order to define an extended transaction model. Reflexion is achieved by adding reflexive software modules that provide a meta interface to the underlying TP monitor, allowing application developers to adjust both the application programming interface and the system functionalities. We propose to use an adaptable transaction manager inspired from the Reflexive Transaction Framework defined by [10]. In the following we introduce this Reflexive Transaction Framework (RTF) and we show that it is sufficient enough to implement our extended transaction model.

The Reflexive Transaction Framework

To accommodate the diversity between different extended transaction models, the RTF introduces a separation of the programming interfaces of the TP monitor. These interfaces are presented in figure 7.

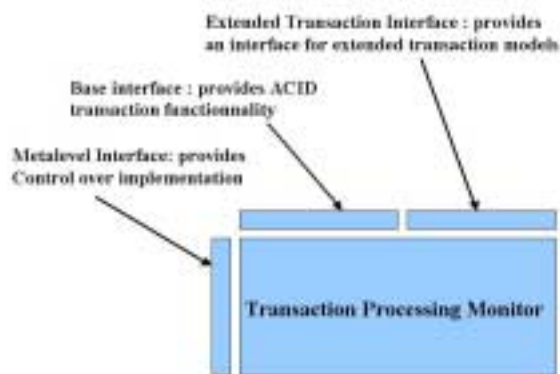


Figure 7: Separation of interfaces to the RTF

The purpose of the meta interface is to modify the behavior, or semantics of the functional interface and the extended transaction interface provides application-level interface with new functionalities.

In the Reflexive Transaction Framework, TP monitor structure and behavior are customized through transaction adapters, which are add-on reflexive software modules that provide the meta interface to control the underlying TP monitor.

A transaction adapter contains a representation of the system; not only are changes in the system reflected in equivalent changes to the representation, but a change in the representation will cause changes in the behaviour of the system. Each adapter corresponds to a particular functional component of the TP monitor, such as transaction execution, lock management, conflict detection and log management. Transaction adapters are composed of a set of meta objects which represent selected behaviors of the underlying functional component, and a meta interface to control the behavior of that component. A modification made to an adapter through the meta interface changes the behavior of the TP monitor. Figure 7 illustrates transaction adapters in the reflexive transaction framework.



Figure 8: Architecture of the reflexive transaction framework

Implementing our transaction model

In our first transaction model, we want to change the isolation property and to provide the new begin operation `BEGIN_R` used to tag reconfiguration transactions. This can be achieved easily as following :

- *Adding the BEGIN_R operator*

Defining this new operator can be achieved by using the transaction manager meta interface. We add a default implementation for this operator as a meta-object inserted in transaction manager adapter. This meta object will tag the transaction as a reconfiguration transaction in the reified data structures of the transaction manager before calling the classic BEGIN operator.

- *Customizing the isolation property*

This can be done by using conflict adapter meta interface to change the conflict detection and to introduce our reconfiguration aware wait die algorithm. In the Reflexive Transaction Framework, this can be achieved by adding a meta object in the conflict adapter in charge of implementing this new behavior. When the lock manager detects a lock conflict between two transaction during a lock request, control is passed to the lock adapter through an upcall, along with all information pertaining to the conflicting request. The lock adapter can then apply our reconfiguration-aware deadlock algorithm to handle the conflict. Figure 9 shows the structure of the reflexive lock adapter that embeds our deadlock prevention algorithm.

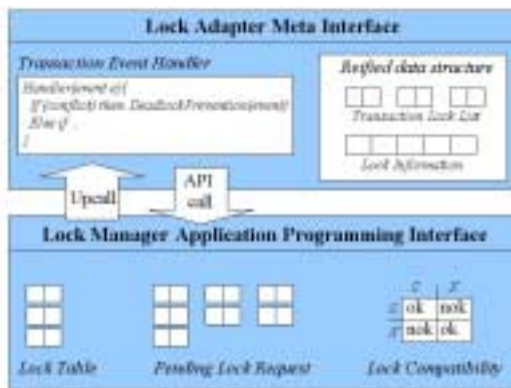


Figure 9 : Structure of a reflexive lock manager

Implementing our reconfiguration algorithm by using the Reflexive Transaction Framework can be done very easily since our reconfiguration solution is defined as an extended transaction model. We can then benefit from all transaction manager frameworks used to customize classical transaction model. Our first solution induces/requires applications to be transactional. This may induce an heavy penalty on application performances.

In the following section we attempt to define another model well suited for non transactional applications and with low penalty on application performance.

4 Discussion

Our previous model fits well with transactional applications by enhancing the standard isolation property. However a lot of applications are not transactional by nature and do not require ACID properties. Such applications does not want to pay full cost of an ACID implementation to be reconfigured

safely. However, in order to be consistent they must have at least the isolation property.

This means that to be reconfigured safely, all applications must be transactional. However the transaction model can be very reduced since the minimal property required is the isolation. As a consequence we plan to define a more general transaction model in which transactions only require a strong isolation property with respect to reconfiguration. We define two kinds of transactions:

- Reconfiguration transactions which run a set of reconfiguration operations. Let RT be the set of transactions of this kind
- Application transactions which only require the isolation property with reconfiguration transactions. Let AT be the set of transactions of this kind.

Each kind of transaction has the following properties:

- Transactions belonging to RT have the classical ACD properties but they are strongly isolated from other reconfiguration transactions and from transactions belonging to AT.
- Transactions belonging to AT have no classical ACID properties but the strong isolation property with transactions belonging to RT. They are not isolated from other transactions belonging to AT.

The required isolation property is the same as defined in 4.2.2 section. Ensuring our strong isolation property in this model is harder to implement than in our previous model since we cannot rollback transactions belonging to AT.

A first way to ensure this property is as follows : When a conflict is detected between a reconfiguration transaction and another transaction, the transaction manager must rollback the reconfiguration transaction. This solution is easy to implement but can lead to reconfiguration starvation.

Another basic solution is based on the fact that a conflict between a reconfiguration transaction and an other running transaction is detected before transactions begin. This approach requires that we can identify statically the set of resources requested by each transaction.

5 Conclusion

Dynamic reconfiguration is a key aspect of distributed application evolution, in terms of adding new functionalities, changing components relationships, and modifying the placement of components in a networked system.

Reconfiguration refers to the modification of an application during execution, while preserving the service availability. Reconfiguration involves the creation, removal or replacement of components, the

modification of interconnections and the migration of components.

The challenge is to provide a dynamic reconfiguration mechanism that maintains application consistency while minimizing the impact on the running application. We have made a distinction between local consistency and global consistency: Local consistency is related to well known problems related to process or component migration and global consistency is related to global application computation. In our mind, a way to identify global calculations is to use transaction demarcations as defined in classical transactional systems.

In a first step, target applications are component-based applications with transactional facilities (such as EJB[1]). With this kind of application, we propose to ensure global consistency by using an extended transaction model with a strong isolation property. Our reconfiguration algorithm benefits from the application transaction model. Reconfiguration orders are applied as a transaction and benefit from classical ACID transaction properties. Reconfiguration is thus reliable despite network or system failure. Furthermore, the reconfiguration algorithm does not have to care about global computation since it is achieved by a transaction manager with special isolation property from other transactions.

We propose to implement this extended transaction model by using an adaptable transaction manager. We plan to define an adaptable transaction manager, inspired from the Reflexive Transaction Framework defined by [10].

Our model fits well with transactional applications by enhancing the standard isolation property. However many applications are not transactional by nature and do not require ACID properties. The remaining work consists in defining a very restricted transaction model which only provides the isolation property to application transactions. We argue that such a transactional model can be even used by non transactional applications since it only adds the cost of isolation. As a consequence, this model can ensure full reconfiguration consistency for a broader kind of applications. We argue that a general solution to ensure reconfiguration consistency relies on the isolation property between different parts of global application computation.

References

[1] V. Matena and M. Hapner. *Enterprise Java Beans Specification v1.1 - Public Draft*. Sun Microsystems, May 1999.

[2] J. Kramer, J. Magee. "The Evolving Philosophers Problem : Dynamic Change Management". *IEEE Transactions on Software Engineering*, pp. 1293-1306, November 1990.

[3] R. Balter, L. Bellissard, F. Boyer, M. Riveill and J.Y. Vion-Dury, "Architecturing and Configuring Distributed Applications with Olan", *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, The Lake District, 15-18 September 1998

[4] BEA et al. *CORBA Components: Joint Revised Submission*. OMG TC Document orbos/99-07-f01..03,0 orbos/99-0f05..07,12,1g. Object Management Group, August 1999.

[5] L. Bellissard, F. Boyer, M. Riveill, and J.-Y. Vion-Dury. "System Services for Distributed Application Configuration," In *Proc. of the 4th IEEE Int'l Conf. on Configurable Distributed Systems, (ICCDs'98)*, Annapolis MD, May 4-6, 1998.

[6] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel", *USENIX Winter Conference*, pp. 283-290, San Francisco, January 1992.

[7] P. Amaral, C. Jacquemont, P. Jensen, R. Lea, and A. Mirowski, "Transparent object migration in COOL-2", *ECOOP*, June 1992.

[8] A. S. Tanenbaum, R. van Renesse, H. van Staveren, and G. J. Sharp, "Experiences with the amoeba distributed operating system", *Communications of the ACM*, December 1990.

[9] D.J. Rozenkrantz, R. E. Stearns, P. M. Lewis, "A system level concurrency control for distributed database systems", *ACM trans on Database Systems*, vol. 3(2), June 1978.

[10] R. Barga, C. Pu, "Reflections on a legacy Transaction Processing Monitor", in *Proc. of Reflection'96*, San Francisco, April 1996.