Globally Consistent Dynamic Reconfiguration

N. De Palma, S. Ben Atallah, S. Bouchenak, D. Hagimont
INRIA Rhône-Alpes, SARDES project
655, Avenue de l'Europe
F-38330 MONTBONNOT SAINT-MARTIN, France
Tel: +33 4 76 61 52 00, Fax: +33 4 76 61 52 52
Email: Daniel.Hagimont@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. In most of the previous works, only local consistency has been addressed, i.e. reconfiguration only ensures consistency of the components state and of the communication channels. In this paper, we argue that local consistency is not sufficient and that consistency of global computations must be ensured. The global computation is generally composed of sessions which should be isolated with respect to reconfiguration. We observed that sessions isolation is similar to transactions isolation, and we therefore propose to use an extended transaction model in order to ensure global consistency when a reconfiguration occurs. We then discuss possible solutions for non transactional applications.

Keywords: distributed applications, components, dynamic reconfiguration

1 Introduction

The growing use of the Internet as an execution environment for distributed applications emphasizes 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 behavior 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 theses 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. In our context, dynamic reconfiguration covers the three 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 version of some components.

A major issue related to dynamic reconfiguration is to ensure that the application remains consistent after a reconfiguration. In the following, we distinguish between two levels of consistency.

With local consistency, reconfiguration ensures consistency of the components state and of the communication channels. However, local consistency allows reconfiguration to occur at any point in the execution of the application.

With global consistency, reconfiguration cannot occur at any execution point. This generally leads to the definition of sessions which are isolated with respect to reconfiguration.

In order to ensure global consistency, we propose a way to capture applicative sessions and to constrain component reconfiguration according to these sessions. 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. In the same vein, we propose to extend a classical transaction model to identify the global calculations (sessions) that must remain consistent upon a reconfiguration.

In a first step, our target applications are transactional applications built as a set of components. We argue that ensuring global consistency with transactional applications can be done easily by using an extended transaction model with an adapted isolation property. Then we discuss possible solution for non transactional applications.

The paper is organized as follows. Section 2 provides an analysis of reconfiguration related problems. Section 3 presents the related work. Section 4 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.

As explained in the introduction, we make a distinction between local consistency and global consistency, respectively discussed in sections 2.1 and 2.2.

2.1 Local consistency

Three basic problems can lead to inconsistencies during reconfiguration. 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.

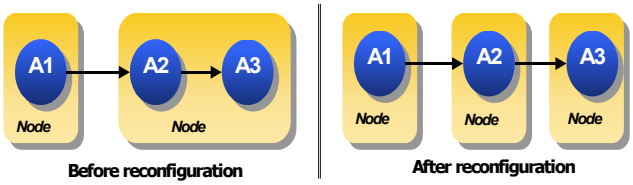

Figure 1 : Migration of A2 from node 2 to node 3

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, as 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 the 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 it execution on the new node.

Status of communication channels

The last issue is related to communication channels: when a component moves to another node, some messages may be in transit and may be lost. In figure 2 we see that when component *A2* moves from node 2 to node 3, m3 is in transit. This message reaches node 2 after *A2* migrated to node 3 and is lost since *A2* is no longer on node 2.

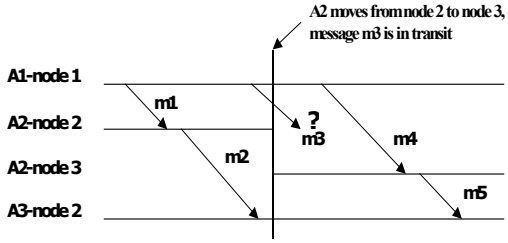

Figure 2 Status of communication channels

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 mechanisms 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 allows component state preservation.
- Java serialization is another technique used to save and restore object states.
- Thread capture mechanisms allow implementing active component migration..

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 techniques 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 [2][3][7][8][10]. 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, local consistency is not sufficient since consistency can also depend on the applications' global calculations, which must remain consistent despite reconfiguration.

The global consistency problem is well illustrated by the example given by [1]. In this paper, we consider the following example. A number of clients nodes *Ci* access a printer server *S* via their agent *Ai* and a server manager node *M* (figure 3). In this case, a calculation may consist of a sequence of message interactions involving *Ci*, *Ai*, *M* and *S*. For instance, *C1* may initiate calculation *s1* to request a print service; completion of *s1* is dependent on the consecutive calculations *r1* and *p*, that *A1* and finally *M* will initiate on *S*, to actually print the data.

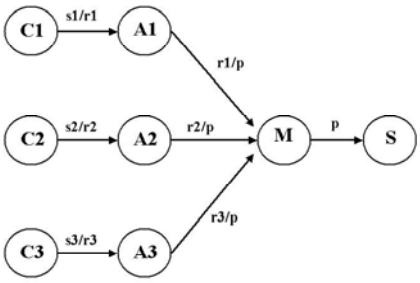

Figure 3 : The printer example

The reconfiguration scenario we consider in this example is the replacement of the printer server *S* by a new version. This reconfiguration scenario involves components *Ci*, *Ai, M* and the printer server *S*. All connections which point to the old printer server must be rebound to the new printer server version (as motivated in the section devoted to local consistency).

Furthermore, in order to ensure global consistency, we cannot replace the printer server by a new version whenever we want. We must take care of the global computations to reconfigure it safely: we cannot reconfigure the application between two calculations *p* since these calculations may belong to more global calculations such as *si*. For example if *Ci* still has further data to send, then *si* is not completed and may require further *ri* and *p* calculations. More practically, it may be inconsistent to reconfigure the printer server in the middle of a printing session which sends data from a client to the printer server.

Ensuring global consistency requires a way to capture global calculations and to constraint component reconfiguration according to this calculations.

3 Related work

Different approaches can be considered for dealing with potential inconsistencies when reconfiguration occurs.

Reaching a reconfiguration point

The Polylith [11][13] environment provides support for the reconfiguration of distributed applications. Applications are designed as a set of modules interconnected through a message bus. Polylith allows module reconfiguration from different points of view: structural, geometric or implementation changes. The management of application consistency relies on the introduction in the source code of primitives for controlling configuration changes. A reconfiguration can only occur at given points called "reconfiguration points" which are defined by the programmer. Reconfiguration actions are ensured to be consistent when a module reaches a reconfiguration point. When a programmer introduces a reconfiguration point, she/he must define how module consistency can be preserved. As a consequence reconfiguration mechanisms are not transparent to programmers and there is no model to help programmers in controlling global calculations.

Reaching an abstract state

Conic [6] is a programming environment based on a component model and a configuration language. CONIC provides support to component addition, deletion and rebinding (i.e. changing the interconnection pattern).

The reconfiguration mechanism focuses on component interactions. At runtime, components are in particular states (called abstract states because they can be deduced from the external observation of their execution). Reconfiguration can only be performed when some components are in a *quiescent state*. In this quiescent state, the state of a target component is ensured to be stable and coherent. The novelty is that a component can be forced to the quiescent state by inhibiting some component interactions. CONIC's reconfiguration only affects component interactions. CONIC ensures global consistency by identifying all nodes belonging to a global calculation and by forcing these nodes to be quiescent. However, reaching the quiescent state induces an important execution disturbance.

Using process migration

Migration provides a way for a process to be moved from one execution site to another during execution. A lot of works [4][[5]14][16] have been done on this topic. Some of these works give a high level of transparency to the programmer in providing generic mechanisms to capture and recreate process states and to deal with communication channel states. Changing applications geometry can be done by using mechanisms such as checkpointing [4] or forwarding techniques [14]. However migration mechanisms do not provide full solutions to dynamic reconfiguration as they do not handle interface and component replacement. Furthermore process migration techniques do not address the problem of global consistency.

This analysis described different ways to achieve reconfiguration. However these solutions do not conveniently address the problem of global calculation. In the following we propose an analyze of the global consistency problem, then we propose a model to identify global calculations and we describe a solution to enforce global consistency despite reconfiguration.

4 Ensuring global consistency

In this section we will analyze 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.

4.1 Analyzing consistency constraint

We propose to capture global calculations by identifying global sessions. For example in the scenario given figure 3, we can identify the global session depicted in figure 4:

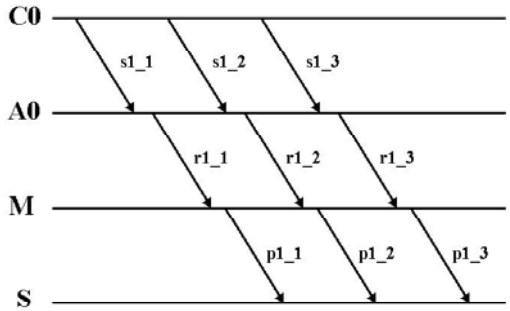

Figure 4: A global session

Ensuring global consistency requires to prevent reconfiguration in the middle of a session, i.e. to isolate global sessions and reconfiguration actions. Then, the question is how to identify global sessions. In our mind, identifying global sessions is similar to transaction demarcations in classical transactional systems. As a consequence, we propose to use an extended transaction model to identify global sessions and to isolate applications sessions and reconfigurations, thus ensuring application consistency in the face of reconfiguration. Assuming that the above session of our example is now modeled as a transaction, reconfiguration constraints must disallow connection modification between the manager node (*M*) and the printer server (*S*) for the time of this transaction.

4.2 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 properties 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 transactions will be consistently applied.

The two kinds of transactions have the following properties:

- Transactions belonging to AT have the classical ACID properties. They are isolated from each other (classical isolation) but they are also isolated (defined below) from transactions belonging to RT.
- Transactions belonging to RT have the classical ACID properties. They are isolated from other reconfiguration transactions and from transactions belonging to AT.

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 transactions and application transactions is the following:

Def2 : 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 ensures 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 an adapted isolation property between different kinds of transactions. This requires to adapt a classical 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 in case of reconfiguration.

Initial wait_die algorithm

Transactions are ordered with timestamps. A transaction is stamped at creation time. 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 before Tj <=> ei<ej
```

When a conflict occurs, the order established between transactions induces either a rollback or a wait for the resource. This method avoids deadlocks 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
```

Reconfiguration-aware Wait-Die

In the following we present an extension of the deadlock prevention algorithm enhanced to ensure our isolation property in case of reconfiguration.

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

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

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 the deadlock algorithm by our reconfiguration-aware algorithm. The following section deals with this issues.

4.3 Technical issues

A way to implement this extended transaction model would be to use an adaptable transaction manager which provides an open implementation of it 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 [12]. In the following we introduce this Reflexive Transaction Framework (RTF) and we show that it is well suited 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 5.

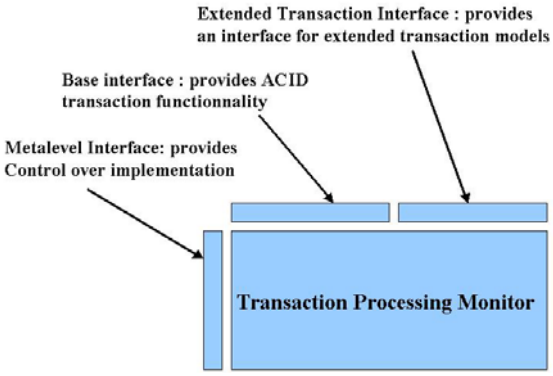

Figure 5: Separation of interfaces in the RTF

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 Reflective Transaction Framework, TP monitor structure and behavior are customized through transaction adapters, which are add-on reflective 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 behavior 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 6 illustrates transaction adapters in the reflexive transaction framework.

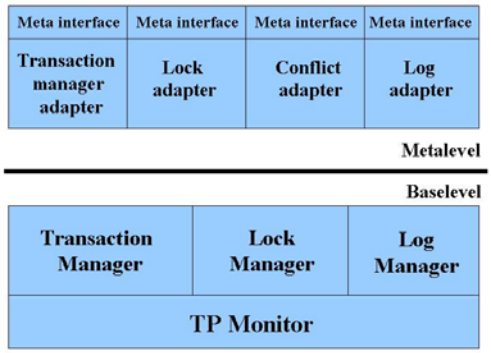

Figure 6: Architecture of the reflexive transaction framework

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 the 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 transactions during a lock request, control is passed to the lock adapter through an upcall, along with all information pertaining to the conflicting requests. The lock adapter can then apply our reconfiguration-aware deadlock algorithm to handle the conflict. Figure 7 shows the structure of the reflexive lock adapter that embeds our deadlock prevention algorithm.

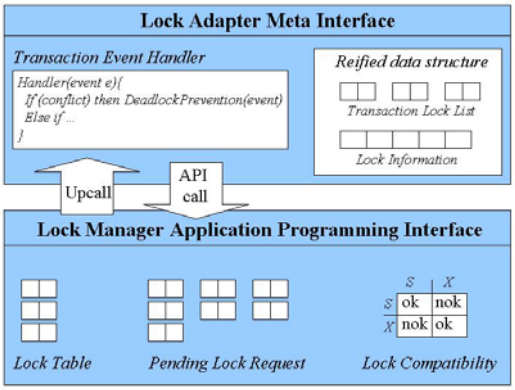

Figure 7: Structure of a reflexive lock manager

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 applications performance.

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

5 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 do not want to pay full cost of ACID properties to be reconfigured safely. However, in order to be consistent they must have at least the isolation property.

The transaction model can be very reduced since the minimal required property is isolation. As a consequence we plan to define a weaker transaction model in which transactions only require an 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 isolated from other reconfiguration transactions and from transactions belonging to AT.
- Transactions belonging to AT have no classical ACID properties but the 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 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.

6 A short comparison with CONIC

In comparison with the CONIC algorithm, our solution has the following advantages :

- Our 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.
- We can easily change the reconfiguration priority by customizing our reconfiguration-aware algorithm thus giving priority to reconfiguration or applicative transactions.
- We are currently conducting an exhaustive performance evaluation to compare our algorithm to the CONIC solution. Intuitively, our algorithm should induce less perturbation on applications performance: in the case of CONIC, all nodes belonging to a calculation must be quiescent. This induces a important execution disturbance since all calculations involved with some of these nodes are locked. In our case, we can use a nested transaction model to allow part of the calculation to run while some nested sub-transactions are locked.

7 Conclusion

Dynamic reconfiguration is a key aspect of distributed applications administration. It refers to the evolution of an application during execution, while preserving the service availability. Reconfiguration can involve 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[15]). With this kind of applications, we propose to ensure global consistency by using an extended transaction model with an adapted isolation property. Our reconfiguration algorithm benefits from the application transaction model. Reconfiguration actions are applied as transactions 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 consistency since it is ensured by a transaction manager with an adapted isolation property.

We propose to implement this extended transaction model by using an adaptable transaction manager. We plan to implement an adaptable transaction manager, inspired from the Reflexive Transaction Framework defined by [12].

Our model fits well with transactional applications, but 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 ensures the isolation property between applications and reconfigurations. We argue that such a transactional model can be 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 class of applications. We argue that a general solution to ensure reconfiguration consistency relies on the isolation property between different parts of the global application computation.

References

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

[2] 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

[3] 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 System*s, (ICCDS'98), Annapolis MD, May 4-6, 1998.

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

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

[6] Kramer J., Magee J., Sloman M., "Constructing Distributed Systems in CONIC", IEEE Trans. Software Engineering, vol.SE-15(N.6), June 1989, pp.663-675.

[7] Shaw M., DeLine R., Klein D. V., Ross T. L., Toung D. M., Zelesnick G., "Abstraction for Software Architecture and Tools to Support Them", IEEE Trans. Software Engineering, vol.SE-21(N.4), April 1995, pp. 314-335.

[8] L. Bellissard, S. BenAtallah, F. Boyer, M. Riveill, "Distributed Application Configuration*", in Proc. 16th International Conference on Distributed Computing Systems (ICDCS'96)*, Hong Kong, April 1996, pp. 579-585.

[9] Raynald M., Schiper A., Toueg S., "The Causal Ordering Abstraction and a Simple Way to Implement It", Information Processing Letters 39(6), 343-350.

[10] DeRemer F., Kron H.H., "Programming-in-the large vs. Programming in the Small", IEEE trans. Software Engineering, vol.SE-2(N.2), June 1976,pp.114-121.

[11] Purtilo J. M., "the POLYLITH software bus", ACM TOPLAS, vol.16(N.1), Jan. 1994, pp.151-174.

[12] Bloom. T., Day M., "Reconfiguration and module replacement in Argus : theory and practice", Software Engineering Journal, March 1993, pp.102-108.

[13] C. Hofmeister, J. Purtilo, "Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for replacement", Proc. 13th International Conference on Distributed Computing Systems, pp. 101-110, 1993.

[14] M. L. Powell and B. P. Miller. Process migration in DEMOS/MP, "Proc.6th ACM Symp. On Operating system Principles", pp 110-119, September 1983.

[15] Sun Microsystems, Enterprise JavaBeans Specifications, Version 2.0, 2001.

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