Livrable Selware SP2

Lot 1, December 21, 2007, 11:14

Selfware Self-Repair Mechanisms and Algorithms

F. Boyer (INRIA)
T. Coupaye (France Telecom)
M. Leger (France Telecom)
S. Sicard (INRIA)

Contents

1 Introduction

Problem determination and repair in complex systems can be cumbersome and costly (due to system unavailability, repair cost, etc.). Self-repair (also called self-healing) is a functional area of autonomic computing defined by IBM [8] which deals with automatic discovery and correction of faults in software systems. Hence a self-healing system must be able to detect problems, diagnose them and recover from faults that might cause some of its parts to malfunction by applying a corrective action, e.g., finding an alternate way of using ressources or reconfiguring the system to keep it working.

A fault model related to the Selfware context has been considered, for instance problems can result from bugs or failures in software or hardwares. A first step is to detect failures resulting from the faults identified in our fault model by using failure detectors and monitoring probes (section 2). The second step is the fault treatment which is composed of fault diagnosis and repair (section 3). The fault diagnosis consists on determining the cause of errors in terms of location and nature. As a result, our autonomic system build a repair plan so as to recover from the error.

2 Failure detection

The taxonomies of faults described in [2] distinguishes the definition of faults from errors and failures. A failure occurs when the delivered service deviates from correct service defined by user requirements. The part of the system which is liable to lead to failure is an error. A fault is the cause (attributed or assumed) of an error in a system. It must be noted that a fault or an error may not necessarily cause a software failure.

2.1 System faults

System faults encompass any kind of faults appearing at the level of the hardware, OS, or middleware supporting the management system. Among these different kinds of faults, we consider more specifically in Selfware the management of fail-stop faults, that lead to a non-responding state a part of the managed system. A fail-stop system either works nominally or do not work at all. Consequently when a fail-stop failure occurs, the failed system never performs erroneous states transformations.

2.1.1 Types of fail-stop failure detectors

Detecting fail-stop failures in a distributed system is a complex task, because in the general case it is impossible to make the distinction between a very slow process from a failed one. The problem is caused by the asynchronism of the communications, meaning that it is impossible to define maximum communication delays between the distributed components. In consequence, most fail-stop failure detectors are considered as weak detectors in the sense that they can make errors when diagnosing a failure (cf. Table 1). A weak failure detector may either see non-existing failures, or it can miss some failures. Two important properties characterize a failure detector regarding this concern: the completeness and the exactness [13]. Completeness is strong when a failure detector never misses failures, while it is weak in the other case. In the same way, exactness is strong in the case a failure

	Strong exactness	Weak exactness	Finally strong exactness	Finally weak exactness
Strong completeness	P	S	$\diamond P$	$\diamond S$

Table 1: Failure Detectors Classification

detector never signals a non-existing failure. Exactness conditions can also be relaxed by considering the temporal dimension. The *finally strong exactness* ensures that, in a finite time, a failure detector will not suspect a correct component.

In practive, failure detectors of type P, S, $\diamond P$, $\diamond S$ are not feasible. The existing failure detectors such as Ping, Heartbeat et Pinpoint (described in the following of this section) only try to approach these properties.

2.1.2 Ping detectors

The Ping failure detector relaxes the hypothesis of asynchronism of the communications by establishing a maximum delay for detecting failed components. This delay estimates the time to go and return for a message in the system including the processing time of this message in a system component. If the delay is I and a component A seeks to determine whether a component B is down, A periodically emits a message to B, which must return an acquittal. If A does not receive an acquittal after a time I, A declares B down. The estimate of I is difficult and not infallible. If the period is too short the risk of false detection is high. By contrast, a cowardly estimate of time delays the detection of failures.

2.1.3 Hearbeat detectors

There are two variations of Heartbeat detectors. The first alternative is similar to a reversed Ping. Its principle is the following: the components emits at a time interval I known to all, a message "I am alive." This message is received by all. If after a period I, since its previous emission component has not reissued the message "I am alive," all the other components regard it as defective. This technique relaxes, as Ping, the asynchronism hypothesis by considering a delay of transmission of messages.

Another variant of the heartbeat detector does not relaxed this hypothesis [10]. In this case, as in the first option, components regularly broadcast a message like "I am living." Here the meantime, separating two programs, does not take into account the delays in transmission, but is the same for everyone. The principle of this sensor is based on a counting of messages between components. Each component maintains a vector for each component involving the number of messages received.

2.2 Software faults

Detection of software faults is more and more difficult as the complexity of software products is increasing. Some common software faults include syntactic, semantic faults, QoS faults. In the classification of Jim Gray [7], software faults are either Bohrbugs or Heisenbugs. Bohrbugs are essentially permanent design faults which are deterministic in nature so they can be identified easily during the early phases of software life cycle. Heisenbugs, on the other hand, belong to the class of temporary internal faults and are intermittent. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproductible (e.g., if the system is restarted).

2.2.1 The Pinpoint detector

Pinpoint is a failure detector based on a behavior analysis system [3], [1]. Failures are detected by observing a variation of behavior. In addition, Pinpoint locates the origin of faults.

In a system consisting of several distributed software components, the behavior of the system (that is, what is going on) is characterized by the interaction between components. It is therefore possible to model the behavior of the system by considering all the interactions among its components and with the outside world. The stakes in the construction of a such a failure detector are twofold. On the one hand, the detector must be able to monitor the interaction between components, and on the other hand he should be able to characterize an observation to be normal or not.

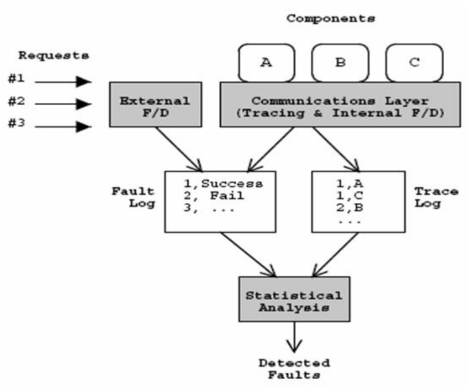

Figure 1: Operating principles of Pinpoint

Figure 1 shows the operating principles of the Pinpoint detector. Monitoring the interactions between components is provided with a specific communication layer, playing the role of an interceptor that is transparent for the components. In this way, all calls are intercepted and taken into account by the detector. The interceptor build logs that are analyzed to extract a representation of the instantaneous behavior of the system.

Figure 2 (left) shows the normal behavior of the application. The normal behavior is achieved through a phase of testing of the system offline. During this phase, an artificial load is injected into the system in conjunction with an injection of faults to infer reactions and the spread of failures in the system. Figure 2 (right) illustrates a case of error detection where the behavior is not consistent with that expected. This detection is based on a statistical test confirming that the detection process does not satisfy the strong exactness property.

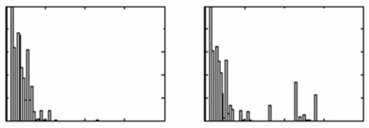

Figure 2: normal vs inconsistent application's behavior

2.2.2 Failures due to software contract violation

Design by contract [11] is a a means to specify the correct behavior of a system with assertions and to check this correctness either statically or at runtime (sometimes only for debugging purpose). The contract must be satified by all parties, e.g., clients and servers in a system. In some cases, a runtime is responsible for the dynamic checking of these assertions. Then a contract violation is a fault which can lead either to a corrective action in the system so as to fulfil the contract (i.e., recovery), or to the renegotiation of the contract, or to a failure. Assertions are of the following forms:

- **Preconditions**: what does it expect?
- **Postconditions**: what does it guarantee?
- **Invariants**: what does it maintain?

Some languages propose these kinds of assertions at the method/procedure level. For instance, the Java Modeling Language (JML) allows to specify contracts in Java regarding :

- acceptable and unacceptable input and output values or types,
- error and exception conditions values or types,
- side effects or side effects free categorization.

The ConFract system [4] follows this paradigm in the Fractal component model, it dynamically builds contracts from specifications at assembly time and updates them according to dynamic reconfigurations, contracts are localized in component interfaces.

Constraint languages are also used in software modelling. OCL [12] is a navigation, query and constraint language for graph-based models, especially for UML models.

```
<definition name="org.objectweb.jasmine.jade.boot.jadenode
 arguments="heartbeat_pulse_period_in_second >
 ...
  <!--          SUB-COMPONENT             -->
  <component name="heartbeat"
    definition="org.objectweb.jasmine.jade.service.heartbeat.HeartBeat">
    <attributes ...
      <attribute name="pulsePeriodInSec"
        value="${heartbeat_pulse_period_in_second}" />
    </attributes>
    <controller desc="primitive" />
  </component>
 ...
```

Figure 3: ADL definition of a Selfware node

2.3 Failure detectors used in the Selfware platform

Two main kinds of failure detectors are currently provided in the Selfware platform, allowing detection of respectively fail-stop system faults and some kinds of transitory applicative faults. Both kind of failure detector can be used to build Autonomic Managers [see SP1-L1].

2.3.1 Fail-stop faults

Fail-stop faults are detected using heartbeat detectors. These detectors are wrapped within Managed Elements (called HeartBeat-ME), which allows deploying and managing them in the same way as any other Managed Elements. In the ADL description of a typical Selfware node, a HeartBeat-ME is declared as a sub-component. This ensures that when a node becomes alive, the Selfware deployment system automatically deploys the HeartBeat detector on it. This deployment process uses the Selfware deployment service described in the [see SP1-L2].

The description given in figure 3 is extracted from a Selfware node's ADL definition. It shows that a HeartBeat detector is defined as a sub-component of a Selfware node. This sub-component has an attribute nammed pulsePeriodInSec, that defines the time interval separating two "I am alive" messages. This attribute's value is given as an instanciation parameter of the Selfware node component.

2.3.2 Failures related to the component model

The Selfware platform relies on the Fractal component model for both the managed applications and the management system. Thanks to the causal connection of reflexive architectures, dynamic reconfigurations in Fractal-based systems can rely on component-based architectures to modify a part of a system during its execution. Reconfigurations may involve every manageable element defined in the component model provided they are reified at runtime, they can be **structural** (e.g., addition or removal of components) or **behavioral** (e.g., lifecycle modifications).

A component model like Fractal describes what the consistency of an application is, especially in terms of architecture and possibly behavior (component interactions) and dynamic reconfigurations should not break this consistency, so we consider here faults due to dynamic reconfigurations of component-based applications. These kinds of faults are related to the violation of modelling constraints mentioned before and are Bohrbugs as they are essentially due to reconfigurations which are not conform to the system specification.

Reliability of dynamic reconfigurations. Runtime modifications may let the system in an inconsistent state i.e. no more available from a functional point of view. Indeed the architecture of the system once reconfigured can be not in conformity with the component model or possibly system specific constraints (e.g. architectural invariants) anymore. For instance adding a component *C1* in another component *C2* could be an invalid reconfiguration if it creates a cycle in the component hierarchy (i.e. if *C2* contains *C1*) which is forbidden by the model. In addition to the consistency defined by the Fractal component model, we want to be able to add application specific constraints, like for instance a structural constraint about the number of subcomponents of a composite component. Therefore we must ensure the conformity of the system to the model and constraints after reconfigurations.

Integrity constraints to define system consistency. In our solution, system consistency relies on integrity constraints expressed both on the component model and on applications. An *integrity constraint* is a predicate on assemblies of architectural elements and component state. Therefore, a system is consistent if all integrity constraints on the system are satisfied. To express constraints including invariants, preconditions and postconditions, we use FPath [5] as a constraint language "à la OCL". Some advantages of FPath are that it can navigate and select Fractal elements at runtime with just introspection capacities. Moreover it is based on a graph representation of the system during execution (cf. figure 4). We distinguish three different levels of constraint specification:

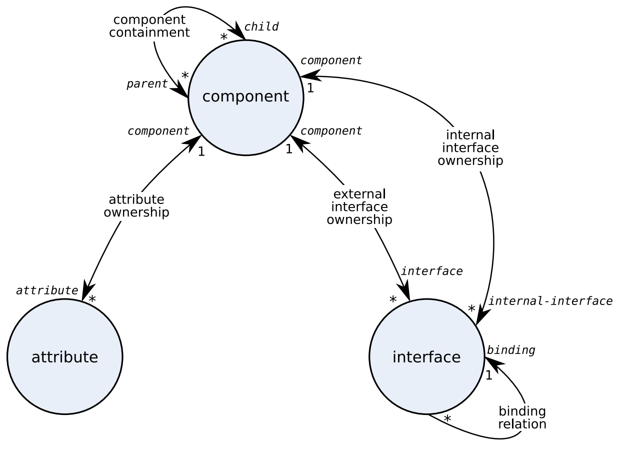

Figure 4: Graph representation of the Fractal component model

- The **model level**: this a set of generic constraints associated to the component model. These constraints apply to all instances of some elements of the Fractal model. Examples of such constraints are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid recursion). We specified the semantics of reconfiguration operations in the model with preconditions and postconditions which should never be violated. An example of such a precondition for the remove operation is to check that all interfaces of the component to remove are unbound:

  ```
  Fractal API: void removeSubComponent(Component child);
  // preconditions:
  not($child/interface::*[bound(.)]);
  ...
  ```

- The **profile level**: this is a set of generic constraints to refine the component model for a family of applications. The profile level conforms to the model level. A profile may for instance forbid component sharing in applications with the constraint `size(./parent::*)<=1` which says that every component cannot have more that one super-component.

- The **application level**: application constraints are specific to a given architecture and apply directly to instances of Fractal elements designed by their names. The application level conforms to the profile and to the model. Invariants can concern cardinality of sub-components in a super-component, two component interfaces which can never be unbound etc. Application constraints are specified in Fractal ADL with the *constraint* tag. For instance, we may want to add a structural invariant on a component *ClientServer* saying that it always contains only one component providing the *service* interface:

  ```
  <definition name="ClientServer">
   ...
   <constraint value="size(./child::*[./interface::service])=1" />
  </definition>
  ```

Constraints can be checked by a dedicated *ConstraintChecker* component both at compile time on a static configuration and at runtime, however only new application constraints can be dynamically added or removed by means of a *ConstraintController*. Pre/post-conditions of primitive intercession operations are checked at each operation execution whereas invariants are only checked at the end of the reconfiguration. That is to say a system can temporarily violate invariants during a reconfiguration but it must be in a correct state after its execution. When it is detected, a constraint violation throw an exception which can be captured by a Recovery Manager.

3 Reparation of Fail-Stop Failures

3.1 Main repair techniques

This section presents the repair technique that is used in the Selfware infrastructure to repair fail-stop system failures. As explained previously, such failures are detected by placing heartbeat detectors on the nodes composing the Selfware infrastructure. The repair technique that is presented in the following sub-sections is based on a simple politic of reparation, that consists in replacing failed components by new ones having the same configuration properties. It should be noted that such technique could also be used for repairing some kinds of applicative faults, in particular those leading components to unrecoverable states, such as memory overflows.

3.1.1 Checkpointing

Checkpoints constitute a classical method for the repair of faults, whether or not distributed [6]. The basic principle is to save the state of the system periodically on a reliable space (eg a disk, or an in-memory reliable space). After a failure, the most recent saved state is restored and execution resumes its course before the failure. The overall state of a distributed system is defined by the union of local states of all processes belonging to the system.

There are two classical types of checkpoints: independent checkpoints and coordinated checkpoints. In the first case, the components of the system carry out their checkpoints independently. After a failure, a synchronization stage is required to restore a coherent global state. This method (optimistic) has the advantage that it involves a minimal additional cost to performances in the absence of failures [15], while the recovery process is complex and may take a certain time. Coordinated checkpoints synchronize all the components to record an overall consistent state. Several approaches are possible to achieve such a backup. A simplistic approach is to block all communications during the recording of the state to ensure that there is no message in transit. This technique is perfectly valid but it creates an important overhead at execution. Accordingly, non-blocking approaches are preferred most of the time. In this case, the challenge is to detect messages from the application emitted during the capture of the state. A well-known example of a non-blocking backup is the "snapshot" of distributed Chandy and Lamport [9].

3.1.2 Logging

Logs rely on the fact that a component can be modeled by an initial state and a sequence of interactions or events. An event may be the receipt of a message or its issuance, a system interrupt, or any other event with an impact on the component. Logging can be done in an optimistic or pessimistic way. Optimistic approaches consider that the failure of a component can occur only when the logging operation is completed. This approach reduces the overhead of process logger. However, if a failure occurs during the logging operation, the repair process is made more complex because of the loss of some part of the log. Pessimistic protocols suppose that a failure can occur after each occurrence of an event. In this type of approach, the log is updated for each event occurrence.

The log is an append-only file written to a stable storage. The main kinds of recovery protocols based on logging in transactional systems are the followings:

- Undo Logging (Write-Ahead Log): once a crash has occured, undo all operations from incomplete transactions. Outputs to disk are done early (before commit).

- Redo Logging: read log from the beginning, redo all updates of commited transactions. Outputs to disk are done late (after commit).
- Undo/Redo Logging: redo all committed transactions (top-down) and undo all uncommited transactions (bottom-up).

3.2 The Selfware repair manager for fail-stop faults

3.2.1 Objectives

The objective is to build a generic repair manager, able to deal with fail-stop failures occurring in any distributed application managed by the Selfware platform. Three main goals can be identified:

- Detecting and repairing fail-stop faults.
- Dealing with nested faults.
- Dealing with faults occurring in the management system itself.

Fault detection is based on the use of failure detectors as presented in section 2. The basics of faults repairing are presented in the following sub-sections. The management of nested faults is considered in a following sub-section, as well as the management of faults occurring in the management system itself.

3.2.2 Default Repair Policy

The repair process is inherently based on the notion of *configuration state* introduced in [ref SP1-L2]. Any managed element in the system exhibit a configuration state, which is composed of the aspects it exposes to the control of a management system. Among the different aspects that can be exposed for a given managed element, some are considered as major for management purposes: its local configuration settings (i.e., its properties values), its life cycle state, and its relationships with the other managed elements. As an example, we can consider the management state of an Apache Httpd server as exposing its current configuration settings (corresponding to properties defined in a file named `httpd.conf`), its life cycle state (started/stopped) and the host address and port of the Tomcat servlet server to which it is connected as a client (if such a connection exists). In case of a failure, the goal of the repair manager is to bring back the failed components to an operational and coherent configuration state. The default repair policy of the Selfware repair manager consists in bringing back these components to their previous configuration state preceding the failure. In the case of the failure of an Apache Httpd server, the repair manager will restart a new Apache Httpd server having the same configuration state as the previous one. This means that the new Apache server will have the same properties defined in its Apache configuration file, the same life cycle state and the same connection with the Tomcat servlet server to which it is connected as a client, if such a connection existed prior to failure.

3.2.3 Design Principles

The repair manager relies on the following two main design principles.

Component-based system A first basic design principle adopted in the Selfware platform is that the overall system, including both the managed application and the management system itself, is represented by a component-based graph. In the case of an application composed of legacy software elements, these elements are associated to components called wrappers, that translate the specific management interface of legacy elements into the uniform management interface defined in the Selfware platform [ref SP1-L2]. After a failure, the repair manager must act on this component graph to localize the failure, to identify its scope in terms of the impacted components, and to repair these components. Figure 16 shows an example of a distributed application managed by Selfware, the overall being represented by a set of components running on three machines.

Using a checkpointing mechanism After a failure, the repair service introspects the component graph in order to get the minimal knowledge allowing the failure to be repaired. The components concerned by the failure are those wrapping the failed node and the software elements which were running on it prior to failure. The repair service must get the configuration state of these components (what are their bindings, their containments, their attributes, etc). To be able to perform this introspection task, these components should be available, even after the failure.

One solution to ensure the availability of these components in case of failures is to forbid a component to be co-located with its managed element. In the case of a failure impacting a managed element, the component wrapping this element is running on another node and may be considered as available despite the failure. However, implying a wrapper to be distant from its managed element is not realistic in the context of multi-tier distributed applications, where the execution time of the management functions can be critical (as for self-sizing services). Moreover, some managed elements (such as nodes) cannot be entirely manipulated from a distant location, implying these to be co-located with their wrapper.

Thus there is a requirement for using a checkpointing mechanism which provides a consistent view of the configuration state of the components. This function is provided by the System Representation Service, that is part of the global Selfware Architecture [ref SP1-L1]. The System Representation service provides a checkpointing of the overall management system, including the managed elements. This checkpointing service should be considered as belonging to the category of independent checkpoints (see 3.1.1), because each component is responsible for checkpointing itself, independently of the other components. Despite this property, the overall state of the checkpoint view is consistent because of the following aspects:

- Only the configuration states of the components are checkpointed (not their business state).
- A sequence of reconfiguration actions can be encapsulated in a transaction, associated with an isolation property ensuring that a component cannot be involved in several reconfigurations at a time.

- The update of the checkpoint view is performed in a synchronous and atomic manner, at the end of each reconfiguration transaction.

Due to the first aspect, the actions that should be taken into account by the checkpointing service are the configuration and reconfigurations actions, acting on components. Because such actions are provided and controlled by the Selfware platform, it is easy to intercept them in order to update the checkpoint view of the components. The second point ensures that in the case two actions a1 and a2 are executed in order on two components $C1$ and $C2$, the checkpoint view cannot reach a state where $C2$ has been updated through $a2$ while $C1$ has not been updated with $a1$. Finally, the isolation property of the transactional system allows to rollback any reconfiguration transaction without raising inconsistencies, in the checkpoint view, between the rollbacked components and the other ones.

3.3 The repair algorithm

The core process executed after detecting a failure involves the following main steps by analyzing the global component graph. We recall that this component graph is always available, through the System Representation service that provides a checkpoint view of this graph.

- **Analysis step**. Identify the failed components and get their management state.
- **Substitution step**. Substitute the failed components identified by the previous step by newly instantiated ones having the same management state.

The general repair algorithm includes the following main steps which are detailed in the following.

Algorithm 1 Global algorithm

Requires: A component graph representing the system to repair.
Ensures: The overall system is repaired in case of a node failure.

```
  Analyse the failure and build a repair plan according to a repair policy
  Clean the global system (remove failed components)
  Execute the repair plan (substitute the failed components by newly ones)
```

3.3.1 Analysing the failure and building a repair plan.

This step consists in analysing the failure and subsequently returning a repair plan, according to a repair policy. It uses the System Representation to introspect the management state of the application prior to the failure, and inserts the references of the components to repair in the repair plan.

Algorithm 2 Analyse the failure & build a repair plan

Requires: *FailedNode* : The reference of the component representing the failed node.
Ensures: *FailedCmps*: The references of the failed components, *RepairPlan*: A repair plan composed of the references of the components to repair (the default policy defines the components to repair as those which were running on the failed node)

```
for all $cmp$ in $FailedNode.getSubComponents()$ do
  $FailedCmps.addCmp((cmp))$
  $RepairPlan.addCmp(cmp)$
end for
```

3.3.2 Cleaning up the global system.

Cleaning up the global system means locating and removing all references from a surviving component to a failed component in the component graph. This is done by using the reconfiguration operations provided by the component model. Because these operations are specialized for each type of legacy element (at the wrapper level), the necessary actions will be automatically performed at the level of the legacy elements (e.g., closing a TCP connection).

3.3.3 Execute the repair plan.

The components to repair have been registered in the repair plan. Each of these components can be introspected to get its configuration state in order to substitute it by a new one having the same configuration state.

3.3.4 Management of nested faults

Nested faults correspond to the case where a fault occurred while a previous one is under reparation. An immediate solution consists in sequencing the management of faults. In this case, the occurrence of a fault 2 is ignored until the management of a previous fault 1 is completed. This approach is valid only in the case the components sets impacted by the faults are disjoints. In contrast, when these components sets overlap, the repair process may failed itself, because some of the components it needs to access to complete the repair of fault 1 have became unavailable due to the occurrence of fault 2. The principle adopted in Selfware to manage nested faults consists in managing priorities. A nested fault is always considered of higher priority than the current fault. In case of the occurrence of a nested fault, the current repair process is interrupted and lost for the benefit of the last fault. The idea is that repairing the last fault will include the repair of the previous ones, because the analysis of the state of the overall management system will detects all the currently failed components, independently of the fault that has caused their failure.

3.3.5 Management of faults occurring in the management system itself

A self-healing repair manager should be able to manage the faults occurring in the management system itself. Ensuring this property is a strong stake of a repair service, highly required for building completely autonomic systems. Reaching this objective implies to

Algorithm 3 Clean up the global system

Requires: $FailedCmps$: The references of the failed components.

Ensures: All relationships (binding, containment) involving a failed component are closed and removed.

```
for all $cmp$ in $FailedCmps$ do
  {for each alive component, remove a failed binding}
  for all $itf$ in $cmp.getServerInterfaces()$ do
    for all $clientItf$ in $itf.getBindingSources()$ do
      if $clientItf.owner()$ not in $FailedCmp$ then
        $clientItf.unbind()$
      end if
    end for
  end for
  {for each alive component, remove a failed son}
  for all $fatherCmp$ in $cmp.getFathers()$ do
    if $fatherCmp$ not in $FailedCmp$ then
      $fatherCmp.removeSubComponent(cmp)$
    end if
  end for
  {for each alive component, remove a failed father}
  for all $cmp$ in $FailedCmp$ do
    for all $subCmp$ in $cmp.getSubComponents()$ do
      if $subCmp$ not in $FailedCmp$ then
        $subCmp.removeFather(cmp)$
      end if
    end for
  end for
end for
```

Algorithm 4 Execute the repair plan

Requires: $RepairPlan$:a repair plan as returned by Algorithm 2, $FailedNode$: the component representing the failed node.

Ensures: The failed components are replaced by newly ones having the same management state

```
1: $newNode = NodeAllocator.replaceNode(FailedNode)$
2: for all $cmp$ in $RepairPlan$ do
3:   {Create an equivalent component (same attributes, interfaces, relationships and life
     cycle state) and deploy it on $newNode$}
4: end for
```

add reliability to both the core process of the repair service, and the critical data it manipulates (i.e. the data provided by the System Representation service). A classical way for adding reliability is to use redundancy. By replicating both the core process of the repair service and the System Representation layer, a fault-tolerant repair service can be provided. However, the number of faults that are tolerated is limited by the replicas cardinality. This is still un-sufficient with regard to a self-healing property, because each time a failure appears in the repair service, a human administrator must detect it and re-establish the replicas cardinality. An interesting principle that is currently under investigation is to make the repair service reflexive, meaning that it should repair itself in case of its failures. In this case, the repair service automatically manages the re-establishment of the replicas cardinality. The basic idea consists in adopting a reflective approach in which the replicas composing the repair service are themselves under the control of the repair service. More concretely, any replica of the repair service is represented in the System representation. In case of a failure of one of these replicas, another replica will detect and repair the failure, which will consequently re-establish the replica cardinality without human intervention.

4 Reparation of Software Faults

4.1 Objectives

We aims to restore the system to the consistent state before the time of failure which can be of the following kinds:

- Software failures: transaction cannot complete due to some internal error conditions. E.g, failures related to the component model due to dynamic reconfigurations, deadlocks.
- System failures: a power failure or other hardware or software failure causes the system to crash.

On the other hand, disk failures (e.g., a head crash or similar disk failure which destroys all or part of disk storage) must prevented by redundancy.

Our solution aims to ensure the reliability of dynamic reconfigurations in the Fractal component model (but it can be generalized to other models), and can be applied to concurrent and distributed reconfigurations. ACID properties in the context of reconfigurations can improve reliability by making systems fault-tolerant i.e., compliant with the specification. These properties are unifying concepts of transactions for distributed computation [14] used for supporting concurrency, recovery, and guaranteeing system consistency. To benefit from these properties, each reconfiguration, defined as a composition of introspection and intercession operations, is executed as a separated transaction called a reconfiguration transaction.

4.2 Transaction properties and model

4.2.1 ACID properties for dynamic reconfigurations

Well-defined transactions associated with the verification of structural and behavioral constraints is a means to guarantee the reliability of reconfigurations i.e., the system stays consistent after reconfigurations. The following definition of the ACID properties in the context of dynamic reconfigurations in component-based systems is given:

Atomicity Either the system is reconfigured and the reconfiguration transaction commits (all the operations forming the transaction are executed) or it is not and the transaction aborts. If a reconfiguration transaction fails, the system comes back in a previous consistent state as if the transaction never started.

Consistency A reconfiguration transaction is a valid transformation of the system state i.e. it takes the considered system from a consistent state to another consistent state. A system is a consistent state if and only if it conforms to our consistency criteria: it does not violate the component model and possibly more specific constraints.

Isolation Reconfiguration transactions are executed as if they were independent. Results of reconfiguration operations inside a non-committed reconfiguration transaction are not visible from other transactions until the transaction commits ; or never if the transaction aborts.

Durability The results of a committed reconfiguration transaction are permanent: once a reconfiguration transaction commits, the new state of the system is persisted so that it can be recovered in case of major failures (e.g., hardware failures).

Each ACID property and associated mechanisms will be detailed further more precisely.

4.2.2 Execution model of reconfiguration transactions

By default, as any reconfiguration could lead the system to an inconsistent state, the execution model of reconfiguration transactions is that every reconfiguration operation must be included inside a transaction: if an operation is executed outside a transaction, a new transaction is automatically created to include this operation. However, this policy can be configured for each reconfiguration operation so that it is not executed in a transaction but then no guarantee is provided that the system will be recovered properly. This last possibility is notably useful when there is no concurrent reconfiguration in the system, so introspection operations cannot do dirty reads and consequently they don't need to be isolated. An explicit language demarcation is necessary when reconfigurations are programmed directly in Java thanks to calls to a *TransactionManager*, otherwise every primitive reconfiguration operation which is not demarcated constitutes itself a single transaction.

Regarding the transaction model, flat transactions appear sufficient for short-lived reconfiguration transactions we consider here. The Two-Phase-Commit protocol [14] is used to coordinate distributed participants in transactions.

4.3 Atomicity of reconfigurations to preserve system consistency

4.3.1 Atomicity of reconfiguration operations

The atomicity property of reconfigurations is notably used to repair transient failures related to the component model. However, this property is also needed to recover from system failures. Operations in Fractal controllers deal with non-functional concerns and constitute primitive reconfiguration operations. We distinguish two kinds of operations belonging to reconfiguration operations:

- **Introspection operations** are without side effects (e.g., the *lookupFc* method in *BindingController* to retrieve the server interfaces which are bound to a given client interface).

- **Intercession operations** modify the system (e.g., the *bindFc* method in *Binding-Controller* to bind two component interfaces).

Reconfiguration can be composed as sequences of intercession operations with conditions expressed by means of introspection operations in component configurations. A classic example of reconfiguration is component hot-swap, i.e replacing an old version of a component by a new one to update an application. In Fractal, this reconfiguration can be implemented with a sequence of several primitive reconfiguration operations of the Fractal API, it implies to *stop* the component, *unbind* all its interfaces, *remove* it from all its parents, *add* the new instantiated component in all the parents, *bind* its interfaces and *start* it (a state transfer operation may be needed if the component is stateful).

4.3.2 A rollback policy to recover from integrity constraints violation

In our approach, to ensure atomicity of reconfiguration transactions, operations performed in transactions which fail because of a constraint violation are undone. Actually, only intercession operations need to be taken into account. When operations are reversible, undoing a reconfiguration transaction is equivalent to sequentially undo primitive intercession operations in reverse order. For every transaction, demarcation and intercession operations are logged in a journal so that it can be undone in case of rollback.

An extensible library of wrappers in Java for primitive operations from the Fractal API is proposed (cf. figure 5) where each intercession operation is associated to its reverse intercession operation in the semantics we choose (typically *bindFc* and *unbindFc* for two component interfaces are inverse operations). Moreover, an operation can keep a state so as to be undone (e.g., changing the value of an attribute requires to keep the old value of this attribute so that the operation can be undone). On the other hand, it is also possible to define non-reversible intercession operations but they need a customized treatment of the rollback (by capturing a *RollbackException* or by listening to transaction events) so as to execute some compensation operations.

4.4 Isolation of reconfigurations to support concurrency

We assume that several administrators may want to reconfigure the same system (potentially distributed) at the same time or a single reconfiguration is itself composed of parallel reconfigurations to optimize the reconfiguration process. There are two available scheduling policies adapted to the level of concurrency for reconfigurations in Fractal applications: either reconfigurations are concurrently executed and accesses to Fractal elements in the system must be synchronized or reconfigurations are serially executed, i.e., only one reconfiguration is executed at a time while others are simply queued.

For concurrency management, we adopt a pessimistic approach with strict Two-Phase locking [14] to provide strong concurrency (corresponding to *serializable* isolation level): locks acquired during a transaction are only released when the transaction is committed to avoid *phantoms* (update lost). Several elements in the model corresponding to FPath

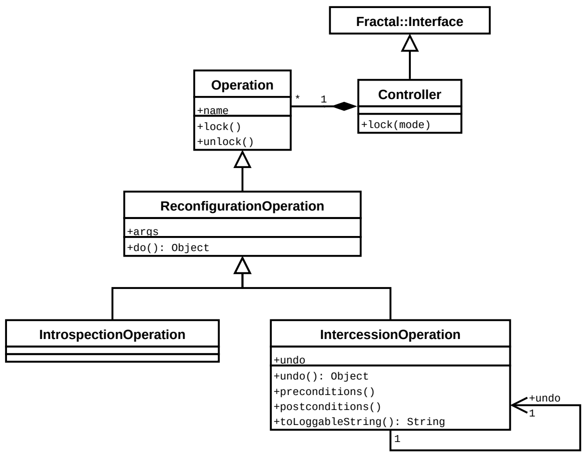

Figure 5: Reconfiguration operation model.

nodes are lockable with a hierarchical algorithm (cf. figure 6): for instance, a lock acquisition on a component for instance will also lock all its interfaces. Locks are reentrant (a transaction can acquire several times the same lock) and there are basically two types of locks: read (or shared) locks, and write (or exclusive) locks. A *LockController* is provided with every component to lock its sub-elements, locks can be also acquired by means of an FPath Expression. For example, we can read lock all subcomponent of the component *ClientServer*: `lock(READ, $cs/child::*);`

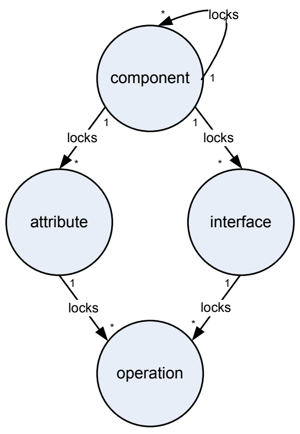

Figure 6: Hierarchical locking model

When a deadlock is detected by the Concurrency Manager in the waiting graph of transactions (i.e. a cycle), a transaction is automatically rollbacked and possibly retried automatically several times with a delay. There are several policies to choose the transaction to rollback: randomly, the last transaction started, etc.

We use this locking model at a finer granularity to avoid inconsistent state for reconfiguration operations based on their semantics. Basically, an introspection operation in a transaction will take read locks on other reconfiguration operations to avoid *dirty reads* and *fuzzy reads* (non repeatable reads): it should not see modifications in the system due to other transactions which have not been committed yet. However, two operations in two different transactions can introspect the same element in the system by sharing a read lock. On the other hand, intercession operations on a given element will take write locks to avoid conflicts between state modifications: other transactions can neither introspect the same element nor modify it. For example, if we want to add the component *Server* in a component *ClientServer*, it will write locks operations in the *ContentController* of the parent and operations in the *SuperController* of the child (*Server*).

4.5 Durability of reconfigurations for catastrophic failure recovery

4.5.1 Recovery protocol

As a recovery protocol for catastrophic failures like a system crash, we use the Undo/Redo protocol with checkpointing (cf. section 3.1). Some assumptions are needed for recovery: failures are detectable and intercession operations are atomic. the recovery protocols implements two actions:

- undo actions: required for atomicity (rollback), rollback is the most common form of recovery.
- redo actions: required for durability (rollforward)

The journal (log) of reconfiguration transactions is kept both in memory and persisted on disk and in case of failure, the journal is redone: all transactions which are not committed are canceled and all committed transactions since the last checkpoint are redone. The log of transactions contains:

- records of every update in the system with the transaction ID. In our case, we do not log system state but rather intercession operations executed in the system. It corresponds to the difference between to consecutive states.
- transaction demarcation events: START, COMMIT, and ROLLBACK
- checkpoint records

The FPath syntax is used in the log format and Fractal elements in arguments of operations are designed with one absolute path (path is not unique due to sharing) in the system architecture.

4.5.2 Component configuration checkpointing

In addition to the journalization of transactions, the system state is periodically checkpointed. We consider here the state of a component-base system as its architecture description and the set of its component state (the only functional state currently captured consists on the set of component attributes). Chekpointing allows to reboot any component in its last know consistent state resulting from a previous successful reconfiguration. The frequency of checkpointing can be adjusted in terms of a number of reconfiguration transactions, it can be basically after each transaction commit.

A component state is checkpointed with Fractal ADL dumps and attribute values are saved either in file or in database. The ADL dumper is extensible so as to consider eventual extensions of the ADL language. State persistence is the responsibility of the *StateController* for each component. Attribute values which are of primitive types are automatically saved in ADL dumps. There are two implementations for other attribute values which are Java Object: either a Java serialization in a file named with an absolute path of the component or it can be saved in a database if an ORM is defined for the attributes of the component. Attributes are tagged as persistent in an extension of the attribute module in Fractal ADL.

References

[1] Research projects in autonomic computing. IBM Research, 2003. http://www.research.ibm.com/autonomic/research/projects.html.

[2] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 01(1):11–33, 2004.

[3] M.Y. Chen, E.Fratkin, E. Kiciman, A. Fox, and E. Brewer. PinPoint: problem Determination in Large, Dynamic Internet Services. In *International Conference on Dependable Systems and Networks (DSN 2002)*, Bethesda (united States), June 2002.

[4] Philippe Collet, Roger Rousseau, Thierry Coupaye, and Nicolas Rivierre. A Contracting System for Hierarchical Components. In *Component-Based Software Engineering, 8th International Symposium (CBSE'2005)*, volume 3489 of *LNCS*, pages 187–202, St-Louis (Missouri), USA, May 2005. Springer Verlag.

[5] Pierre-Charles David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceedings of the 5th Fractal Workshop at ECOOP 2006*, Nantes, France, July 2006.

[6] E.N. (Mootaz) Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[7] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[8] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer Magazine*, 36(1), 2003.

[9] L. Lamport K.M. Chandy. Distributed Snapshots: Determining Global States of Distributed Systems. In *ACM Transactions on Computer Systems*, February 1985.

[10] S. Toueg M. Kawazoe Aguilera, W. Chen. Using the HeartBeat Failure Detector for Quiescent Reliable Communication and Consensus in Partionable Networks. *Theorical Computer Science*, 22, 2003.

[11] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[12] OCL 2.0 Specification. *http://www.omg.org/docs/ptc/05-06-06.pdf*, 2005.

[13] S. Toueg T.D. Chandra, V. Hadzilacos. The Weakest Failure Detector for Solving Consensus. In *11th Annual ACM Symposium on Principles of Distributed Computing*, 1992.

[14] Irving L. Traiger, Jim Gray, Cesare A. Galtieri, and Bruce G. Lindsay. Transactions and consistency in distributed database systems. *ACM Trans. Database Syst.*, 7(3):323–342, 1982.

[15] W.K. Fuchs Y.M. Wang. Lazy Checkpointing Coordination for Bounding Rollback Propagation. In *Symposium on Reliable Distributed Systems*, 1993.