Towards Reliable Distributed Reconfiguration

Ngoc-Tho Huynh
Institut Telecom
Telecom Bretagne
Brest, France
ngoc.huynh@telecom-bretagne.eu

An Phung-Khac
Institut Telecom
Telecom Bretagne
Brest, France
an.phungkhac@telecom-bretagne.eu

Maria-Teresa Segarra
Institut Telecom
Telecom Bretagne
Brest, France
mt.segarra@telecom-bretagne.eu

ABSTRACT

In component-based software engineering, reconfiguration often refers to the activity of changing a running software system at the component level. Reconfiguration is widely used for evolving and adapting software systems that cannot be shut down for update. However, in distributed systems, supporting reconfiguration is a challenging task since a reconfiguration consists of distributed reconfiguration actions that need to be coordinated. Particularly, this task becomes much more challenging in the context of unstable networks where nodes may disconnect frequently, even during reconfiguration. To address this challenge, we propose a platform supporting distributed reconfiguration that includes a solution for managing system states at reconfiguration time. We define (1) different system states regarding reconfiguration and (2) ways that the system will act accordingly. When a disconnection is detected during a reconfiguration, the system may correct reconfiguration plans to continue the reconfiguration if possible, or recover if the reconfiguration fails.

Keywords

Component-based Software Engineering, Distributed Reconfiguration, Reliability

1. INTRODUCTION

In Component-Based Software Engineering (CBSE), reconfiguration often refers to the activity of changing a running software system at the component level. Such changes rely on underlying component platforms that provide reflection facilities. Based on these facilities, a reflective application can dynamically reconfigure its own internal component architecture to adapt to operating conditions. In other words, these facilities support intercession ability of the reflective application. This component-based reconfiguration approach allows raising an abstraction level that is higher than the source code one where software designers operate. This approach promotes using architectural models of the running application to reconfigure it, thus supporting introspection ability of the reflective application and reducing the complexity of the application development.

Particularly, software reconfiguration is strongly related to the domain of runtime software evolution and adaptation. In this domain, reconfiguration is used as a means for evolving and adapting software systems that cannot be shutdown for update. Through a reconfiguration, the application moves between so-called source and target variants. Reconfiguration actions include component removals and additions; interfaces connections and disconnections; and functional calls to stop and start components. On the other hand, successful reconfiguration must enable the target variant to be initialized with data of the source one. Reconfiguration actions thus also include function calls for launching operations to transfer data between the variants. Particularly, in distributed software systems where distributed components usually collaborate each others, changing a component on one site may require to change other components on other sites. Therefore, distributed reconfiguration actions are usually highly dependent.

In distributed systems, supporting reconfiguration is a challenging task when a reconfiguration consists of distributed reconfiguration actions that need to be coordinated - a reconfiguration action on a component may be required by others actions on others components. Particularly, this task becomes much more challenging in the context of unstable networks where nodes may disconnect frequently, even during reconfiguration. These disconnections during reconfiguration may impact the consistency of the reconfiguration actions, the source variant, and/or the target variant. Therefore, the reflective application must be able to correct reconfiguration actions, or in the worse case, recover from failed reconfiguration.

To address this challenge, we propose an architecture of reflective distributed applications called reflective cloud component (RfCC). This architecture is based on an underlying platform called reflective platform cloud component (RpCC) that includes a solution for managing system states at reconfiguration time. We define (1) different system states regarding reconfiguration and (2) ways that the system will act accordingly. When a disconnection is detected during a reconfiguration, the system may correct reconfiguration plans to continue the reconfiguration if possible, or recover if the reconfiguration fails.

The paper is organized as follows. Section 2 includes a motivating example. Section 3 presents basic concepts related to the platform. Section 4 describes the platform ar-
architecture. Section 5 describes a solution towards reliable reconfiguration in the platform. Section 6 discusses related work. Section 7 concludes the paper and anticipates future work.

2. MOTIVATING EXAMPLE

Consider an application used for sharing files over network, called the FS application. Users of this application include some sharers and an admin. The sharers insert files into the system, search for files according to the properties they are interested in (such as file type and keywords), and get these files. In addition to the activities of a sharer, the admin can also view the total number of shared files and remove files according to the file identifiers.

There are various ways to implement this application. Shared data (i.e., files and properties) may be centralized on one server, distributed on some servers with or without replication, or distributed on users devices that are considered as peers of a peer-to-peer network. In the case where the data are distributed, different distributed algorithm (such as Chord or Pastry) may be used to localize these data. On the other hand, different abstract data type such as tree or list may be used to implement these data.

Each implementation variant mentioned above has its own non-functional properties and may be suited to different operating conditions. Therefore, the FS application may adapt its implementation to operating conditions by reconfiguring its running variant to obtain another one. This reconfiguration at runtime enables to reduce the deployment cost and improve the availability of the application. However, it is easily impacted when a node leaves the application during reconfiguration time. The remainder of this paper presents a platform in which in problem is tackled.

3. BASIC CONCEPTS

This section presents basic concepts including cloud component (CC), reflective CC (RfCC), and reflective platform CC (RpCC).

3.1 CC: Component-as-a-Cloud

CC is a specific concept dedicated to developing distributed software systems, originates from medium. It allows simplifying the system specifications by raising a high-abstraction level where the distributed environments do not need to be considered. At this level, a CC is specified similarly to ordinary components. That is, it is “a unit of composition with contractually specified interfaces” and “is subject to composition by third parties”. On the other hand, at the implementation level, the CC is a logical aggregation of distributed components called managers.

4. RECONFIGURATION PLATFORM

This section describes theRpCC architecture and explains reconfiguration principles.

4.1 Model-based Reconfiguration

The RpCC reconfigure the FcCC using models of the FcCC resulted from a specific development process. This process considers the FcCC as a software product lines, and then, allows deriving different implementation variants of the FcCC as member software products. The process also results

- architectural models of the variants, each one describes the component structure of a variant and architectural constraints (in term of elements’ multiplicities);
- a feature model describing the variability of these variants;
• a reconfiguration constraints model containing deployment contracts of the variants; and

• a data transfer model annotating operations for manipulating data in every variants.

At runtime, plans for reconfiguring the running FcCC variant to obtain another one are automatically generated using these models.

4.2 Distributed Reconfiguration

We consider a distributed reconfiguration in an RdCC as a global reconfiguration process composed of distributed local reconfiguration processes, one per RpM instance (Figure 3).

We consider a distributed reconfiguration process as a global reconfiguration process composed of distributed local reconfiguration processes. A local reconfiguration process is executed using a local reconfiguration plan (Figure 3). Each local process is composed of reconfiguration actions separated into successive reconfiguration steps. We identify 7 steps: (1) stop CC services, (2) add new components, (3) start new components, (4) transfer data from to-be-removed components to new-added ones, (5) stop to-be-removed components, (6) remove components and connect new-added components, and (7) restart CC services. Every step is synchronized (described by synchronization barriers in the figure). Within each step, a reconfiguration action in a local process may require another action in another local process. Each local process is executed using a local reconfiguration plan.

4.3 Component Structure

Figure 4 includes the RpC component structure (light-grey part). This structure is resulted from a layered architecture of the RpC. Particularly, a RpC is separated into three layers including communication, CC logic, and reconfiguration. These layers are designed as CCs including CmCC, LoCC, and RcCC, respectively. An RpC is thus composed of three managers including CmM, LoM, and RcM.

First, the CmCC provides a communication layer for other CCs. It is implemented based on JGroups[4], a reliable group communication system implemented in Java. JGroups enables to send and receive point-to-point and point-to-group messages. It also allows detecting and notifying about joined/left-crashed nodes. Furthermore, every node maintains a common “view” being an ordered list of all the nodes, thus allowing to dynamically choose a coordinator node (the first one in the list). These facilities are particularly suitable for implementing targeted features of the RfCC such as coordinating distributed actions and managing the RfCC states. Second, the LoCC provides a logic layer based on which an FcCC instance can understand the global architecture of the FcCC. It manages the architectural model of the FcCC running implementation variant. It is implemented on top of the CmCC. Third, the RcCC implements reconfiguration activities including planning and execution, and the coordination of these activities on distributed FcCC instances. To this end, it uses services of both the CmCC and the LoCC, and manages development models. An RcC is therefore composed of subcomponents including a planning manager (PM), an execution manager (EM), and a coordination manager (CM).

The RpCC can be considered as a reflective middleware for deploying FcCC. An FcM instance of an FcCC is deployed and runs on top of an RpM instance. It uses services provided by the LoM instance and is reconfigured by the RcC instance. FcM instances may communicate each others directly or via the CmCC.

All the components described above are implemented using the Julia implementation (in Java) of the Fractal component model [4]. This hierarchical and reflective component platform is particularly appropriate to our approach. As a Fractal component, each FcM provides a “technical” interface called ContentController. This interface enables to introspect and reconfigure the internal structure of the FcM. In addition, Java Reflection facilities are used to execute data transfer actions.

5. Towards Reliable Reconfiguration

In this section proposes a solution towards reliable reconfiguration in RpCC. We describes the state management and its implementation.

5.1 State Management

5.1.1 Reconfiguration States

Figure 5 describes reconfiguration states of the RfCC including running, reconfiguring, reconfiguration pending and waiting. The RfCC state changes when events such as “bad leave”, “good join” occur or actions such “correct plan”, “rollback” are finished.
Figure 5: State diagram of reconfiguration cloud component

In state **running**, the FcCC of the RfCC is servicing properly. If there is a bad leave (i.e., the architectural constraints of the running variant are violated), the state changes to **waiting** (for the “good join”) where the FcCC services are not available. When the RpCC receives a reconfiguration command, the FcCC is inactivated and the RfCC state changes to **reconfiguring** once a quiescent state is obtained. In this state, if there is a “bad leave” (i.e., a disconnection which impacts reconfiguration plans), the state will become **reconfiguration pending**. The RpCC verifies architectural constraints of the source and the target variants, and the current step of reconfiguration. It determines to rollback or to correct plans. There are the following rules:

- A variant is inconsistent if its architectural constraints are not satisfied (e.g., the architecture requires “at least two servers”, but only one is available)
- The reconfiguration can not be continued if the target variant is inconsistent
- Data can not be transferred (i.e., step 4 can not be performed) if the source variant is inconsistent
- Data transfer can not be rolled back (i.e., step 4 can not be undone) if the target variant is inconsistent
- State **running** can not recovered if the source variant is inconsistent

Based on these rules, we separate 5 reconfiguration pending” substates and define the transitions (Figure 5).

5.1.2 Correcting Reconfiguration Plan

As mentioned above, the reconfiguration plan can be corrected in order to continue the reconfiguration. Particularly, the reconfiguration plans are impacted when the local plan related to the disconnected node contains actions being required by or requiring other actions. In this case, to correct reconfiguration plans, the EM at each node removes constraints related to the disconnected node.

5.1.3 Recover from Failed Reconfiguration

In order to rollback, the coordinator will generate “reconfiguration reverse plans”. Reverse steps include (1) create and connect to-be-removed components (for the reconfiguration which is now failed) (2) start to-be-removed component, (3) transfer data back to the source variant, (4) stop new-added component, (5) remove new-added component, and (6) restart CC services (if the constraints are satisfied). Reconfiguration step 7 (restart CC services after reconfiguration) is never reversed. Reverse plan may contains several reverse steps, depending on the current state of the reconfiguration.

5.2 Implementation

Figure 6 includes a sequence diagram describing the interaction between components of the RpCC within a reconfiguration.

**Figure 6: Reconfiguration cloud component sequence diagram**

First, the CM at the coordinator node (called **coordinator CM**) receives a reconfiguration command. It asks the connected PM to generate local reconfiguration plans and sends them to all the EM (called **local EMs**) via the local CMs.

Next, the coordinator CM sends commands to local EMs (also via the local CMs) to sequentially perform the 7 steps previously presented. A local EM performs the reconfiguration steps by executing the actions described in the local plan. If an action is required by other ones, notifications will be sent to all the other related local EMs once the action is done. Inversely, if it requires other actions, the local process is blocked until all the necessary notifications are received.

If a node is disconnected during the reconfiguration, the JGroup-based CmCC sends notifications to all the LoMs. Each LoM forwards the received notification to the EMs.

Particularly, if the disconnected node is not the coordinator, the selected CM will decide how to act, i.e., correct
Another approach is proposed in [8] to ensure reliable reconfigurations. This approach has been defined four ACID properties in the context of dynamic reconfigurations in component based systems. To ensure atomicity of reconfiguration transactions, operations performed in transactions which abort are undone. All operations are logged in a journal so that it can be undone in the case rollback. In addition, system consistency relies on integrity constraints both on the component model and on applications and FPath have been used as a constraint language.

In [7], the authors presented an end-to-end solution to define and execute reliable dynamic reconsideration of open component-based systems while guaranteeing their continuity of service. It uses a multi-stage approach in order to deal with the different kinds of possible errors in the most appropriate way; in particular, the goal is to detect errors as early as possible to minimize their impact on the target system. A solution relies on a multi-stage validation chain with two main dependability methods: fault prevention and fault tolerance. Fault prevention notably includes the use of static analysis on a dedicated reconfiguration language in order to detect invalid reconsideration with respect to the architecture model, and an additional simulation stage on the target architecture. Fault-tolerance is ensured by a transactional runtime for the actual execution of reconsideration.

Generally, the existing approaches provide solutions for (1) reconfiguration in non-distributed systems or (2) reconfiguration in distributed systems but not “distributed reconfiguration” which is composed of multiple distributed process. In our approach, since we adopt a specific architecture (CC) of distributed application, we can deal with some aspects such as coordinating distributed actions and acting against inconsistency of distributed plans due to node disconnections. On the other hand, our approach leverages using models to perform reconfigurations.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have dealt with some specific problem in support distributed reconfiguration. Supporting this kind of reconfiguration is difficult since such a reconfiguration is composed of distributed reconfiguration process containing inter-processes dependencies between their reconfiguration actions. Furthermore, under unstable network conditions, node disconnections during reconfiguration make these reconfiguration actions become inconsistent. We presented a distributed reconfiguration platform that includes a solution for managing system states at reconfiguration time. We define (1) different system states regarding reconfiguration and (2) ways that the system will act accordingly. When a disconnection is detected during a reconfiguration, the system may correct reconfiguration plans to continue the reconfiguration if possible, or recover if the reconfiguration fails.

Our future work includes applying the approach to appropriate practical use cases in order to better validate the approach.

8. REFERENCES


[2] N. Bencomo and G. Blair. Using architecture models to support the generation and operation of


