Run-Time Root Cause Analysis in Adaptive Distributed Systems

Amit Raj, Stephen Barrett, and Siobhan Clarke

School of Computer Science and Statistics
Trinity College of Dublin, Ireland
{araj,Stephen.Barrett,Siobhan.Clarke}@scss.tcd.ie

Abstract. In a distributed environment, several components collaborate with each other to cater a complex functionality. Adaptation in distributed systems is one of the emerging trends that re-configures itself through components addition/removal/update, to cope up with faults. Components are generally inter-dependent, thus a fault propagates from one component to another. Existing root cause analysis techniques generally create a static faults' dependencies graph to identify the root fault. However, these dependencies keep on changing with adaptations that makes design-time fault dependencies invalid at run-time. This paper describes the problem of deriving causal relationships of faults in adaptive distributed systems. Then, presents a statechart-based solution that statically identifies the sequence of methods execution to derive the causal relationships of faults at run-time. The approach is evaluated, and found that it is highly scalable and time efficient that can be used to reduce the Mean Time To Recover (MTTR) of a distributed system.

Keywords: Distributed Systems, Root cause analysis, Fault causal relationship, adaptive system, component-based system.

1 Introduction

In pervasive computing, the adaptive distributed systems (ADS) adapt (add/remove/update components) themselves to meet the changing functional and non-functional requirements both (say location, bandwidth, QoS, SLA, etc.,). Such a system is fault prone as it may assemble large number of un-foreseen components at run-time that may cause a number of run-time faults such as interface mismatch, component failed, service failed, etc. As components are generally interdependent, a fault in one component may propagate and cause another fault in another component. Thus, the original fault becomes manifest as a different fault [1]. The symptoms or evidence of such a fault may not contain the sufficient information to discover the root cause fault.

Existing root cause analysis (RCA) techniques generally use a graph to analyze the faults causal relationships. When a fault causes another fault, such faults are referred as causally connected faults and represented in a faults' causal-connection graph (FCG). We define FCG as a directed graph $G = (V, E)$, where

Y.T. Demey and H. Panetto (Eds.): OTM 2013 Workshops, LNCS 8186, pp. 292–301, 2013.

V is a set of faults that will be represented as nodes and E is the set of causal relationships among faults that will be represented as edges. Our FCG mainly aims to represent inter-component faults causal relationships. In an ADS, the construction of such an FCG is non-trivial primarily due to two reasons. Firstly, FCG constructed at design-time becomes invalid at run-time due to components adaptation. Thus, a run-time FCG construction mechanism is required. Secondly, in an ADS that is constituted dynamically through plug-in/out of components at run-time, there is no opportunity to apply existing FCG construction techniques such as simulations, testing, fault injections, statical analysis, etc. A few run-time techniques exist such as failure path analysis, execution path analysis, log analysis [2] [3] [4] [2], but they drastically increase the MTTR.

2 State of the Art

RCA approaches can be categorized into two categories: non-dependency based and dependency-based approaches. In ADS, RCA techniques generally require dependency-based approaches [3] [5] [2], thus we focus our review on this category, which are further subdivided into design-time and run-time techniques.

2.1 Dependency-Based RCA Approaches

Dependency-based approaches generally use the topology of components or dependency relationship between faults. These approaches include codebook approach, fault propagation/causal graphs and active probing. The dependency-based approaches generally use a graph, for example, codebook approach uses a directed graph that has faults and evidence events as the nodes, fault propagation/causal graph approaches use a directed graph having faults as nodes, and active probing uses more specific graphs such Bayesian network.

Design-time FCG Construction for RCA. The design time techniques perform simulations and tests on design-time available components to analyze the causal relationships between faults to construct an FCG. Candea et al. [4] and Le et al. [6] illustrate an automatic failure-path generation technique. They declare two faults that lie on a same execution path as causally connected. Candea et al. [4] technique takes no prior information, but takes hours to run and require several reboots that is not feasible for a highly available ADS.

Andrews et al. [7] describe the use of directed graph of component's variables to construct the fault tree where faults are linked with logical operators. This approach assumes a static set of components, whereas the set is likely to change after an adaptation at run-time. Prescott et al. [8] describe the fault propagation analysis using petri-net representation of a system, where fault propagation is observed through relevant tokens that propagate across the petri-net model. It constructs an FCG only when a symptom is identified that will do a reactive RCA and increases the MTTR. Liu et al. [5] describe a fault diagnosis mechanism that uses a fault dependency relationship matrix. However, the derivation of

matrix was not described in the paper. Moreover, in a dynamic system, such matrices are required to be re-generated after each adaptation that becomes a bottleneck in the performance. Most design-time techniques carry out fault correlation through simulations, test cases, deployment information or static analysis. However, the limitation to foresee the run-time situations, limits the use of design-time generated FCG.

Run-time FCG Construction for RCA. The run-time fault causal relationship detection techniques either tag the request to analyze the visited components [2], analyze the logs [3], or analyze the execution paths identified after each request response cycle and incrementally create the graph [4]. For example, Bellur et al. [3] discover the call traces and aggregate them to generate the topology graph of components and construct a Bayesian network of faults. However, it does not explain how to derive a causal relationship between two faults. Similarly, Lo et al. [9] create a Bayesian network where nodes are the components and edges represent the causal connection between components. When a fault evidence is detected, the Bayesian network is analyzed to locate the faulty component. However, the mechanism does not identify the specific fault in the faulty component.

Huang et al. [10] describe a passive event correlation technique that correlates the symptoms with faults. However, until the symptoms occur, faults cannot be analyzed. Similarly, Yemini et al. [11] describe an event correlation technique in real-time. They consider a set of symptoms as a code that is decoded to analyze the problem. The technique uses the topology of the system to create a bipartite graph that is required to generate the code. In an ADS, the topology of the system changes, thus bipartite graph changes and hence the code is required to be regenerated that may become bottleneck in performance. Ensel [12] describes an approach for automatically creating service dependency in a service-based system. Chen et al. [2] identify a fault propagation path by tagging a request and tracking it while traversing through components. It identifies the failed and successful component for a specific request id. Based on the success and failure status of a component for a specific request, it identifies a component that has the closest proximity for the existence of a fault. It helps in identifying the faulty component, however further analysis will be required to find out the exact root cause of a fault in the faulty component.

The discussed techniques cannot derive faults causal relationships to construct/update an FCG at run-time. This is because (1) Non-dependency based approaches do not construct an FCG, and (2) Dependency-based approaches cannot work because:

3 Problem

The research problem is illustrated through our case study that is an extended version of a smart office scenario described by Morin et al. [13]. The smart office uses an ADS that is a Message Transmission System (MTS, see Figure 1), to facilitate the communication between office employees. When a boss wants to

send meeting details to one of his employees, the message is transmitted through MTS. The MTS adapts itself according to the context change and delivers a message at a suitable endpoint. For example, a voice message is delivered to employee's smart phone when he is driving, whereas an email message is delivered when he is at the office desk, assuming that the office does not allow the use of smart phones in their premises. The adaptations in MTS occur according to changes in the context variables. An environment may have several context variables, however, for brevity, only two context variables have been chosen: (1) employee location, and (2) internet bandwidth.

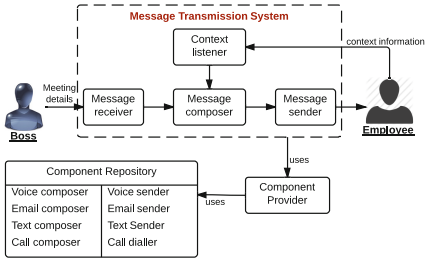

Fig. 1. Message Transmission System

Table 1. Adaptation Matrix

Scenario	Employee location	Internet Bandwidth	Message composer	Message sender
1	Driving	High	Voice composer	Voice sender
2	Office Desk	High	Email composer	Email sender
3	Cafeteria	High	Call composer	Call dialer
4	*	Low	Text composer	Text sender

When the context variables' values changes, the MTS imports the required components from Component Repository (CR) and deploys into the system at run-time to provide the required functionality. The MTS requests the Component Provider (CP) to provide the requested component. The CP connects to the CR, take out the required component that will be deployed into the MTS. Similar practices have been seen in dynamic discovery of services using UDDI registry [14]. A possible set of adaptation scenarios is described in Table 1.

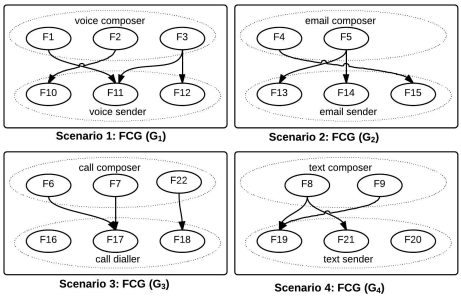

Fig. 2. Varying FCG for different scenarios

The MTS dynamically deploys and un-deploys several components at run-time. An analysis of faults in such components was carried out. Due to space constraint, detailed faults were not described, but they are labeled as f_1, f_2 ... f_{22}. The required FCGs for different scenarios are described in Figure 2. Assuming

these scenarios will occur in a sequence (1→2→3→4), an existing FCG will become invalid for the current scenario. In scenario 1, a fresh FCG (G_1) was created, but in scenario 2, the FCG (G_1) becomes invalid and a new FCG (G_2) is required. Similarly, in scenario 3, G_2 becomes invalid requiring G_3, and in scenario 4, G_3 becomes invalid and needs G_4.

The first step in constructing an FCG is to detect the causal relationship between faults. The causal relationship in an FCG can be broadly categorized into two categories: (1) Intra-component, and (2) Inter-component. Intra-component fault causal relationship detection does not require information about run-time coworker components. Thus, such relationships in a component can be analyzed in isolation from rest of the system at design-time using existing techniques [4] [6] [8].

However, the inter-component faults causal relationship detection is a challenging task primarily because it requires information about run-time coworker components. The key problem is to proactively derive faults causal relationships, in a dynamically constituted ADS, among faults of different components to construct/update an FCG after an adaptation.

4 Approach

The paper illustrates a state-based approach to determine the causal relationship between two faults. Our approach is based on the following hypothesis.

4.1 Hypothesis

Several fault correlation techniques (discussed in section 2) illustrate that two faults are causally connected if they fall in same execution path [6]. An execution path is a sequence of methods execution. When two methods lie on an execution path, the faults of these methods will be inter-dependent. Given this information, our hypothesis states that if two methods are in a sequence of execution, then the faults of preceded method are causally connected with the faults of succeeded method.

4.2 Part-Whole Statechart

The Part-Whole Statechart (PWS) [15] [16] describes the notion of a system-wide statechart (Whole) and components statecharts (Part or PC or PSC). In the Whole statechart, a transition from one state to another happens when an activity occurs whose statechart is described by a Part statechart. The PWS mechanism is described in Figure 3. The state transition in Whole statechart is carried out through an event whose functionality is implemented by a component. When the component statechart is executed and reached its final state, the component functionality is completed, and at that instant, the Whole statechart makes a transition to a next state.

In order to derive the fault causal relationships, the Part statecharts are further analyzed. We are using voicemail sending scenario of MTS, but the following

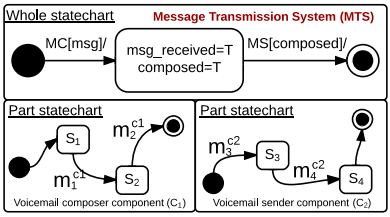

Fig. 3. A detailed PWS of MTS for voicemail sending scenario

Algorithm 1. State-based fault correlation Algorithm

Input: Whole statechart (W), the set of Part statecharts (P= $\{P_1 ... P_n\}$) where n is the number of components, and FCG.
Output: FCG

```
state ← start state of W
Q ← Breadth First Search Queue for W
Q.enqueue(state)
while Q ≠ empty do
  state ← Q.dequeue()
  {E} ← set of all outgoing transition events from state
  for each e in E do
    P_e ← Part statechart for e
    {M^ce} ← set of methods making transition to final state in P_e
    {F^ce} ← set of faults for all method m ∈ M^ce
    nextstate ← getDestination_State_Of_Event(e)
    Q.enqueue(nextstate)
    {E_next} ← set of all outgoing transition events from nextstate
    for each e_next in {E_next} do
      P_e_next ← Part statechart for e_next
      {M^ce_next} ← set of methods making transition from start state in P_e_next
      {F^ce_next} ← set of faults for all method m ∈ M^ce_next
      Node n_src ← FCG.createNodeIfNotExist(F^ce)
      Node n_dest ← FCG.createNodeIfNotExist(F^ce_next)
      FCG.makeEdgeBetween(n_src,n_dest)
    end for
  end for
end while
```

mechanism is generic and can be applied to any scenario of an ADS. In the Whole statechart, the first event is the message compose 'MC' event. This event will be carried out by voicemail composer component. In a Java based component, a task is carried out by executing a set of Java class methods. Therefore, the state transitions in a Java based component will happen through its class methods. In Figure 3, the method m_2^{c1} finishes the voicemail composition functionality and makes a transition to the final state of its Part statechart, and then, the Whole statechart reaches the next state. When the voicemail is composed, the Whole statechart executes the message sending event 'MS'. In the voicemail

sender component, the method m_3^{c2} makes a transition from start state to S_3 state. Following the sequence of events, it is clear that the methods m_2^{c1} and m_3^{c2} are in a sequence where later one executes next to the former one. According to the hypothesis (section 4.1), it can be established that the faults of method m_2^{c1} are causally connected with the faults of m_3^{c2} using Algorithm 1. Now, the fault correlation will have two cases:

- **Case 1:** When a method has only one fault, say m_2^{c1} has f_{21}^{c1} and m_3^{c2} has f_{31}^{c2}, it can be derived that the faults f_{21}^{c1} is causally connected with f_{31}^{c2}.
- **Case 2:** When methods have more than one fault, say m_2^{c1} has f_{21}^{c1}, f_{22}^{c1} and m_3^{c2} has f_{31}^{c2}, f_{32}^{c2}, the causal connection between specific faults cannot be derived. The causal connection can be derived between fault sets such as the fault set $\{f_{21}^{c1}, f_{22}^{c1}\}$ is casually connected with the fault set $\{f_{31}^{c2}, f_{32}^{c2}\}$. Thus, a further investigation will be required to derive fault correlations between specific faults.

4.3 Results

As we are the partners of EU project "TRANSFoRm" [17], we applied this approach in the project. It is a distributed system that is composed of several components situated across different geographical locations. In order to collaborate with different partner's components, we implemented the system-wide workflow in Apache Camel. The components, developed by the partners, may add/delete/update dynamically in the workflow. This property is similar to that mentioned in section 3. Thus, we choose "TRANSFoRm" project as a good use-case.

In the "TRANSFoRm" project, different components provide their functionality through web-services. We build components as OSGi bundles that are running on OSGi platform to support adaptation. The owner of each component supplied the component's Part Statechart, and Whole statechart was built from the workflow written in Apache Camel. We applied our approach and found inter-component faults correlation graph as shown in Figure 4.

4.4 Limitations

Our approach is limited to find the correlation between faults arise due to functional requirements violations. However, in highly critical domains, our analysis is likely to be not sufficient for the faults occur due to violations of non-functional requirements.

5 Evaluation

5.1 Scalability

Scalability is one of the important factor to assess the applicability of a mechanism in todays distributed systems. Here, scalability mean efficiency of our algorithm against the number of components in a system. In order to assess

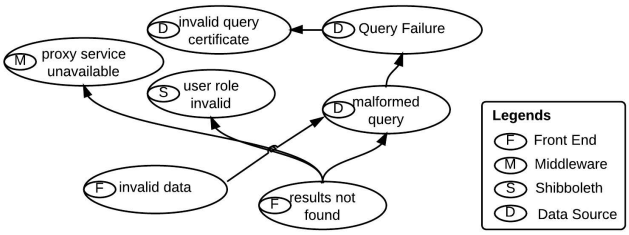

Fig. 4. A correlation between faults

the scalability of our mechanism, we ran several experiments on Java Virtual Machine having a heap memory of 1024 MB. As our mechanism requires only Whole and Part state charts as input (Algorithm 1), we developed a Java-based agent that automatically creates a Whole statechart and several Part statecharts based on a given domain theory [18]. An extremely large-scale system may have thousands or millions of components, thus, our experiments ran over a system having 100 to 1000,000 PSCs. The results are described in Figure 5. The time taken in case of 1000,000 PSCs was high, thus we ran our algorithm on 2 and 3 Java threads separately. We found a tremendous downfall in the time taken as shown in Figure 6. It is worth noting that the time taken was extremely low because we managed to keep a list of pointers in a statechart that directly points to the states that have direct transition to final states. Thus, our statecharts can be defined as a quintuple $< E, S, s_0, \delta, F, \phi >$ where E is a set of input alphabet, S is the set of states, s_0 is the initial state, $\delta : S \times E \times S$ is the state transition function and F is the set of final states, ϕ is the set of states that have direct transitions to final states F. It avoids the traversal time to find out the required states and increases time efficiency. It reduces the time for root cause analysis that reduces MTTR.

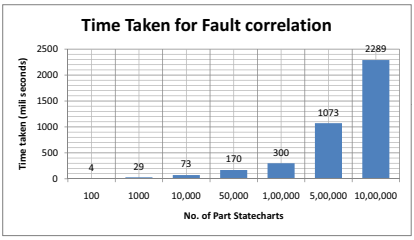

Fig. 5. Scalability Analysis for fault correlation time taken in an ADS

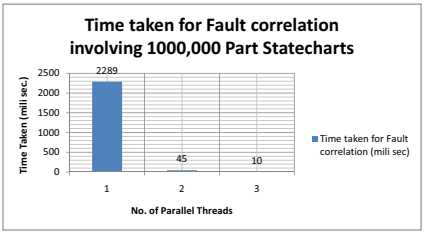

Fig. 6. Scalability Analysis for fault correlation time taken in an ADS having 1000,000 PWCs

5.2 Parallelism

A fault correlation between three faults, i.e., $f_1 \rightarrow f_2 \rightarrow f_3$ can be divided into two pairs of faults relationship, i.e., $f_1 \rightarrow f_2$ and $f_2 \rightarrow f_3$ (assuming transitive relationship between faults). In order to detect such pairs of faults, only two PSCs are required: one corresponding to preceding transition of a state in Whole statechart (preceding PSC), and second corresponding to succeeding transition of the same state in the Whole statechart (succeeding PSC). A pair of preceding and succeeding PSCs can be analyzed independently to detect fault correlation between two faults (as described in Algorithm 1). It is the property of our mechanism that makes our algorithm parallel. Thus, it can be scaled for an extremely large-scale system.

6 Conclusion

The paper describes the problem of fault correlation in adaptive distributed systems and presents a statechart-based solution. In an adaptive distributed system, a component is likely to dynamically install/update/remove in a system. In such cases, fault correlations are changed with every adaptation that makes an existing fault correlation graph invalid after an adaptation.

The paper illustrates that current techniques for fault correlation are not suitable as (1) either they work at design-time and cannot foresee run-time situations, or (2) they work at run-time that increases the MTTR. We presented a state-chart based run-time technique to analyze the fault correlations to support the root cause analysis in adaptive distributed systems. The paper presents the results of our experiment and evaluated our approach for scalability. We found that our approach is highly scalable and can be utilized in large-scale system involving thousands of components.

Acknowledgments. This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Engineering Research Centre (www.lero.ie).

References

1. Abdelmoez, W., Nassar, D., Shereshevsky, M., Gradetsky, N., Gunnalan, R., Ammar, H., Yu, B., Mili, A.: Error propagation in software architectures. In: Software Metrics. In: Proceedings of 10th International Symposium on Software Metrics, pp. 384–393 (September 2004)
2. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings of the 2002 International Conference on Dependable Systems and Networks, DSN 2002, pp. 595–604. IEEE Computer Society, Washington, DC (2002)
3. Bellur, U., Agrawal, A.: Root cause isolation for self healing in j2ee environments. In: Proceedings of the First International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2007, pp. 324–327. IEEE Computer Society, Washington, DC (2007)

4. Candea, G., Delgado, M., Chen, M., Fox, A.: Automatic failure-path inference: A generic introspection technique for internet applications. In: Proceedings of the The Third IEEE Workshop on Internet Applications, WIAPP 2003, p. 132. IEEE Computer Society, Washington, DC (2003)
5. Liu, Y., Ma, L., Huang, S.: Construct fault diagnosis model based on fault dependency relationship matrix. In: Proceedings of the 2009 Pacific-Asia Conference on Circuits, Communications and Systems, PACCS 2009, pp. 318–321. IEEE Computer Society, Washington, DC (2009)
6. Le, W., Soffa, M.L.: Path-based fault correlations. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 307–316. ACM, New York (2010)
7. Andrews, J., Brennan, G.: Application of the digraph method of fault tree construction to a complex control configuration. Reliability Engineering and System Safety 28(3), 357–384 (1990)
8. Remenyte-Prescott, R., Andrews, J.: Modeling fault propagation in phased mission systems using petri nets. In: 2011 Proceedings - Annual Reliability and Maintainability Symposium (RAMS), pp. 1–6 (January 2011)
9. Lo, C.H., Wong, Y.K., Rad, A.B.: Bond graph based bayesian network for fault diagnosis. Appl. Soft Comput. 11(1), 1208–1212 (2011)
10. Huang, X., Zou, S., Wang, W., Cheng, S.: Fault management for internet services: Modeling and algorithms. In: IEEE International Conference on Communications, ICC 2006, vol. 2, pp. 854–859 (June 2006)
11. Yemini, S., Kliger, S., Mozes, E., Yemini, Y., Ohsie, D.: High speed and robust event correlation. IEEE Communications Magazine 34(5), 82–90 (1996)
12. Ensel, C.: Automated generation of dependency models for service management. In: Workshop of the OpenView University Association, OVUA 1999 (1999)
13. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ run.time to support dynamic adaptation. Computer 42, 44–51 (2009)
14. Walsh, A.E. (ed.): Uddi, Soap, and Wsdl: The Web Services Specification Reference Book. Prentice Hall Professional Technical Reference (2002)
15. Pazzi, L.: Part-whole statecharts for the explicit representation of compound behaviours. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 541–555. Springer, Heidelberg (2000)
16. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. 8(3), 231–274 (1987)
17. 7th Framework Programme European Commision: Transform project (April 2013), `http://www.transformproject.eu/`
18. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: Proceedings of the 22nd International Conference on Software Engineering, ICSE 2000, pp. 314–323. ACM, New York (2000)