ICST Transactions on eBusiness Research Article ICST.ORG

The Implementation of A Dependency Matrix-based QoS Diagnosis Support in SOA Middleware*

Jing Zhang, Xiaoqi Zhang, Yi-Chin Chang and Kwei-Jay Lin*

Department of Electrical Engineering and Computer Science, University of California Irvine,Irvine, CA 92697-2625, USA

Abstract

When an SOA business process fails to deliver the desired quality of service (QoS), it is necessary to identify the faulty services that cause the problem since the source of the problem may not be at where the problem is observed. In this paper, we propose a polynomial time diagnosis algorithm by using a dependency matrix for business process structure in SOA. The dependency matrix is built based only on process workflow structure, with no need for historical knowledge on prior executionx. By comparing the performance data reported from business process probes, the proposed diagnosis algorithm also checks some predicates-on-probes (PoP) to increase the monitoring and diagnosis accuracy. We have implemented the diagnosis support for the dependency matrix based QoS management in the Llama middleware. A performance study using some realistic services running on networked Web servers shows that the system can achieve a diagnosis completeness of up to 80%.

Keywords: SOA, Middleware, QoS, Diagnosis

Received on 05 May 2010; accepted on 05 January 2012; published on 05 September 2012

doi:10.4108/trans.eb.2012.e2

1. Introduction

Service oriented architecture (SOA) provides an easy-to-use and flexible paradigm for integrating distributed services into business processes (BP) [3, 15]. Using SOA, enterprise systems can discover and compose services from different service providers both statically or dynamically. However, the flexible and dynamic composition must be carefully managed to ensure the BP service quality. In many applications implemented using SOA, it is important to have an efficient mechanism to identify and replace faulty services that cause a quality of service (QoS) problem in a BP execution.

To detect QoS violations and to identify the cause of failures, SOA middleware must monitor individual service behaviors at run-time. However, in a large SOA system, the monitoring and debugging cost could be a burden to the system. If run-time data are collected and inspected at all services, it may impose significant overheads. In our research, we therefore design an SOA middleware to monitor only a subset of services in a BP at run time [22]. Whenever an end-to-end QoS problem is detected at a monitored service, the middleware triggers a diagnosis algorithm to identify possible faulty services. Assuming that QoS violations do not happen often, we believe this approach is more efficient than monitoring all services all the time.

The Llama middleware project [11, 23] proposes an SOA accountability framework to provide a QoS-driven service composition and management solution that can *detect*, *diagnose* and *defuse* QoS faults in SOA. Such a middleware benefits SOA users by making business processes configurable, traceable and repairable[23].

There are two main issues for QoS diagnosis in SOA. One is the complexity issue since multiple-fault diagnosis for systems with a large number of components is known to be computationally challenging. Most multiple-fault diagnosis algorithms for model-based reasoning [8, 9] have an exponential time complexity. Similar complexity issues are encountered using probabilistic methods such as Bayesian networks, which is NP-hard [6]. Since our goal is to conduct diagnosis at run-time to improve end-to-end QoS, a high complexity is not desirable for the scalability of SOA systems.

The other issue on diagnosis is that both model based reasoning and probabilistic methods require domain knowledge or historical data on prior process executions. But since SOA is a dynamic paradigm where BP may be composed on demand to adapt to fluid requirements, expert knowledge or historical data about a process may be hard

*Corresponding author. zhangjing00@gmail.com

to come by. Lack of process knowledge will reduce the effectiveness of both diagnosis methods.

In [21], we have designed a polynomial-time QoS diagnosis method based on the dependency matrix model [18]. A dependency matrix is constructed using only the BP flow structure that can be easily captured from business process definitions such as BPEL. We have designed an efficient multi-fault diagnosis algorithm for SOA. The diagnosis algorithm utilizes *predicate-on-probes* (PoP) so that it can identify fault locations more accurately. This paper extends the work in [21] and presents a practical system implementation with performance study.

The contribution of the work reported in this paper includes:

- We have implemented the diagnosis algorithm and system support as part of the Llama SOA middleware [11], including service response time monitoring, agent communication, and diagnosis engine.
- We have designed the verification mechanism in the middleware to resolve the ambiguous faulty services problem, which is the major weakness of dependency matrix-based diagnosis.
- We have measured the performance of the diagnosis middleware using realistic but controlled services deployed on networked servers.

The rest of this paper is organized as follows. A QoS accountability framework is presented in Section 2. Section 3.1 shows how to generate the dependency matrices: CDM and PDM. Section 3.2 presents how to select evidence channels (EC), and Section 3.3 shows the diagnosis algorithm. Section 4 defines *predicate on probes* (PoP) and shows how to generate and use PoP. The system design and implementation is presented in Section 5. Section 6 presents the system performance study. Section 7 reviews some related work. Finally, the paper is concluded in Section 8.

2. QoS Accountability Framework in SOA

An SOA BP consists of multiple services and can be modeled as a work flow $G = (V, E)$. Every vertex in V represents an atomic service and every edge in E represents an interaction between two services. In this paper, we make the following assumptions on business processes:

1. A business process is a Directed Acyclic Graph (DAG); i.e. for any service s, there is no non-empty directed path starting and ending on s in the flow.
2. Atomic services may be faulty, but network communications between services are error-free.

Figure 1 shows a PrintAndMail BP. There are 17 services, running on three networked servers deployed on the Llama middleware [11] which has an *Accountability Authority (AA)* and multiple *Agents* that are used to monitor and inspect run-time performance of individual services. When an intermediate end-to-end QoS violation is detected in a BP, an agent sends monitored data to AA, which then performs fault diagnoses to identify likely fault(s). More on Llama will be discussed in Sec. 5.

In a business process, every service deployed on Enterprise Service Bus (ESB) can be used to collect execution information and acts as a potential BP *probe*. Every service has one output probe and at least one input probe. For example, in Fig. 1, service SelectVendors has five input probes and one output probe. For efficiency reasons, not all services in a system are monitored and analyzed at run time. Zhang et al. [22] propose *evidence channels* (ECs) to monitor a subset of services and present two algorithms for evidence channel selection: k-median and set-covering. For example, in Figure 1, BulkMailQuoteB, SelectVendors, AssemblePressSheet, and BeginBulkMailing are selected as evidence channels. If a service is selected as an evidence channel, data can be collected from its output probe and all of its input probes. In Fig. 1, service SelectVendors is an evidence channel; all its input probes and output probe can be used to collect run-time data.

An exception condition reported by a probe may be resulted from the QoS violation of any preceding service in the BP. Our work uses a dependency matrix (discussed in the next section) to describe the relationship between probed results and service states. In general, all predecessor services before a probe are potential causes for QoS violation detected by that probe. In Figure 1, services AddressQuoteA, AddressQuoteB and BulkMailQuoteB can be the causes of exception at BulkMailQuoteB's output probe.

3. Dependency Matrix Based Diagnosis

In the proposed diagnosis framework, different algorithms are used at different SOA deployment phases. At pre-runtime, a system generates the dependency matrix for a business process. The dependency matrix is then used for selecting evidence channels, modeled as a set-covering problem. At runtime, QoS data are collected from selected ECs. The status of probes on selected ECs is analyzed; if some probe shows QoS violations, the diagnosis procedure will be triggered. In this section, we present our design for each of the phases.

3.1. Dependency Matrix

A dependency matrix defines the relationship between probes and services. In a dependency matrix D, D_{ij} denotes the relationship between $probe\ i$ and $service\ j$. D_{ij}=1 means that service j may be a fault source for an exception condition detected on probe i; otherwise D_{ij}=0. In a *complete dependency matrix* (CDM), for every probe, all its predecessor services in the BP's structure are set as potential sources.

Given a BP's service graph, Algorithm 1 identifies the probes and builds the CDM for the BP. Every service s has six tuples <*id*, *parent*, *next*, *pred*, *beginT*, *finishT*>. *id*

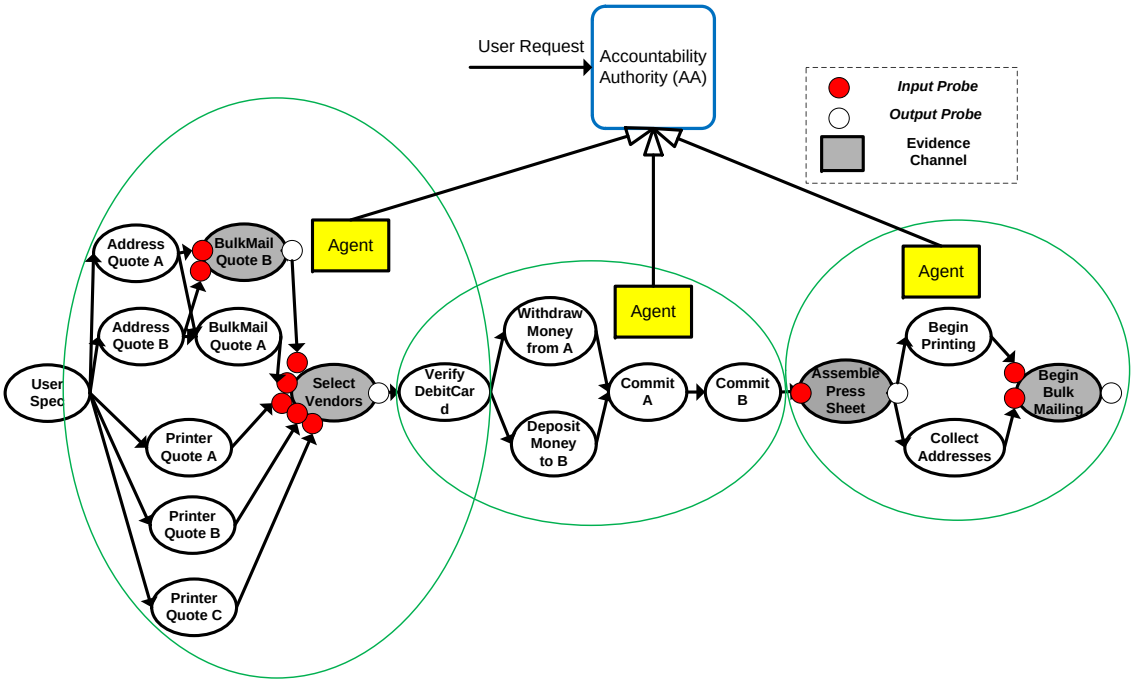

Figure 1. PrintAndMail Scenario in QoS Accountability Framework

is the service identifier. *parent* is the list of its immediate predecessor services. *next* is the list of direct successor services. *pred* is the list of all ancestors of the service. *beginT* is the start time of the service, which is equal to the latest finish time of all its parent services. *finishT* is the expected completion time of the service.

Every probe p has four tuples <*PID*, *EC*, *Type*, *Path*>. *EC* is the service that the probe is on. *Type* denotes whether it is an input or output of the service. *Path* is identified by its parent service ID on the input path.

In Algorithm 1, for every service n (line 3), input probes are created and recorded on lines 3-10 and output probes on lines 11-16. Suppose the number of services is n, the complexity to process one probe (lines 4-8, 11-15) is $O(n)$. Given the probe number p, the complexity of Algorithm 1 is $O(pn)$.

However, QoS violation in a faulty service may be masked by other services between the faulty service and a deployed probe. For example, in Fig. 1, suppose the expected response time for every service is 5 seconds. If service AddressQuoteB takes 7 seconds to finish and all other services run for 2 seconds, the input probe of service SelectVendors will not be able to detect an exception at AddressQuoteB since the end-to-end response time for AddressQuoteA, AddressQuoteB, BulkMailQuoteA, and BulkMailQuoteB together is 9 seconds and causes no response time violation.

To avoid this problem, we modify CDM so that the dependency relationship is limited to only a subset of predecessor services. We define a *partial dependency matrix* (PDM) so that a probe is used to detect the BP response time violation within a sub-window of the end-to-end response time window. PDM is generated from CDM given a time window size. If a service s is a predecessor of the probe and the time interval between the begin time of s and the time of the probe position is less than the defined time window, then s is considered to be a source for the probe. Otherwise, s is not a potential source of the probe exception.

Figure 2 shows a subProcess of PrintAndMail, the original CDM of this process is shown in Table 1 by black entries. Suppose that the response time for every service is x and the time window for PDM is also x, the (0) entries mark the changes from CDM to PDM for Fig. 2.

3.2. Evidence Channel Selection (ECS)

In Sec. 3.1, dependencies between service faults and probe exceptions are discussed. A service's coverage is defined as the union of all its probes. The ECS problem can be mapped to the set covering problem [7] as follows:

- Input: (1) U, a total of m elements to be covered; (2) several sets S_1, ...S_n; (3) the capability of S_i, i.e., elements in U that are covered by S_i. Note that some elements can be covered by more than one sets.

- Output: a subset of S_1, ...S_n that (1) cover all elements in U; and (2) the number of selected sets is minimized.

The set covering problem is NP-Hard. The Regular Greedy Algorithm (RGA) is one of the best approximation algorithms [7] for solving the set covering problem. ECS using the set covering model is shown in Algorithm 2. The

Table 1. CDM (PDM) for Process in Fig. 2

Probe	AdQA	AdQB	BMQA	BMQB	PQA	PQB	PQC	SelV
0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	1 (0)	1 (0)	0	1	0	0	0	0
5	1	0	0	0	0	0	0	0
6	0	1	0	0	0	0	0	0
7	1 (0)	1 (0)	1	0	0	0	0	0
8	0	0	0	0	1	0	0	0
9	0	0	0	0	0	1	0	0
10	0	0	0	0	0	0	1	0
11	1 (0)	1 (0)	0	1	0	0	0	0
12	1 (0)	1 (0)	1	0	0	0	0	0
13	0	0	0	0	1	0	0	0
14	0	0	0	0	0	1	0	0
15	0	0	0	0	0	0	1	0
16	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1

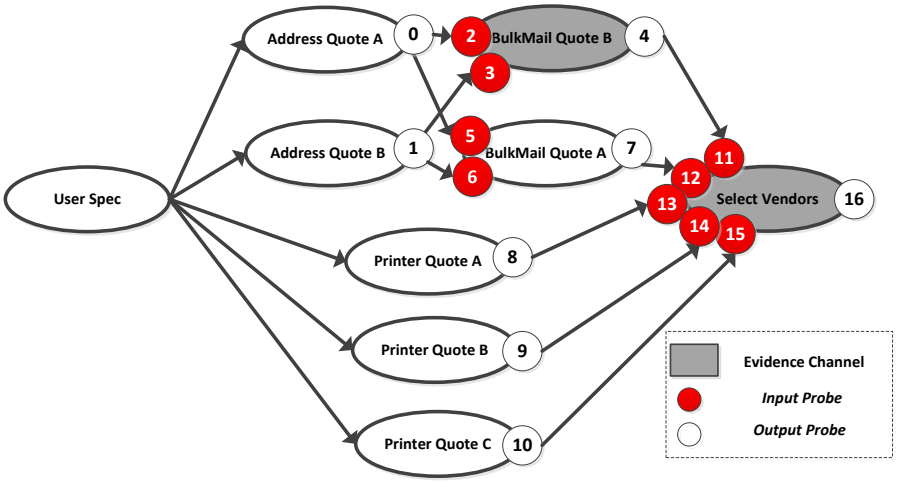

Figure 2. Select Vendors Sub-Process in PrintAndMail

basic idea is that at each step it selects the evidence channel that covers the largest number of services.

In Algorithm 2, every service s's coverage $C[s]$ is initialized on lines 5-11. The complexity of this part is $O(pn)$, where p is the probe number and n is the service number. The critical step of the greedy EC selection (line 12-18) is finding the service with the largest coverage and updating the coverage. Time complexity of finding the service with the largest coverage is $O(n)$ and it will be executed at most n times, so the total time complexity is $O(n^2)$. When EC selection completes, every service's coverage set should be empty. There are at most n^2 services in those coverage sets in total, so the time complexity for updating coverage is $O(n^2)$. In summary, the time complexity is $O(pn + n^2)$.

For example, in Fig. 2, service SelectVendors is selected as EC in the first round and service BulkMailQuoteB is selected as EC in the second round. After these two rounds, all services in the process are covered by ECs. Table 2 shows the dependency matrix for two selected ECs, services BulkMailQuoteB and SelectVendors.

3.3. Dependency Matrix Based Diagnosis

Rish et al. [18] propose a dependency matrix based multi-fault diagnosis algorithm for distributed network. An important difference between our QoS diagnosis for SOA and [18] is that a probe in our scheme does not always detect a faulty service, since a slow service behavior may be masked by other fast services before probe detection. Considering the uncertainty of fault detection and in order to capture as many faulty services as possible, we need to modify the diagnosis algorithm for QoS violation.

Table 2. PDM for Fig. 2 after EC Selection

Probe	AdQA	AdQB	BMQA	BMQB	PQA	PQB	PQC	SelV
2	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0
11	0	0	0	1	0	0	0	0
12	0	0	1	0	0	0	0	0
13	0	0	0	0	1	0	0	0
14	0	0	0	0	0	1	0	0
15	0	0	0	0	0	0	1	0
16	0	0	0	0	0	0	0	1

Algorithm 1 CDM Generation Algorithm

Input: service list N in a business process
Output:CDM, probe list PL.

1: $\forall i, j$, set $CDM[i][j] = 0$
2: set $PID = 0$
3: **for all** $n \in N$ **do**
4: **for all** $p \in n.parent$ **do**
5: set $PL[PID] =< PID, n.id, "input", p.id >$
6: **for all** $s \in (p.pred \cup p)$ **do**
7: set $CDM[PID][s.id] = 1$
8: **end for**
9: $PID = PID + 1$
10: **end for**
11: set $PL[PID] =< PID, n.id, "output", null >$
12: **for all** $s \in (n.pred \cup n)$ **do**
13: set $CDM[PID][s.id] = 1$
14: **end for**
15: $PID = PID + 1$
16: **end for**

Algorithm 2 Evidence Channel Selection Algorithm

Input: PDM; Probe List PL; Service set S=$\{s\}$.
Output: Selected EC SEC.

1: set $SEC = \emptyset$
2: **for all** $s \in S$ **do**
3: set coverage set $C[s] = \emptyset$
4: **end for**
5: **for** $i \in PL$ **do**
6: **for all** service $s \in S$ **do**
7: **if** $PDM[i][s.id] = 1$ **then**
8: add s to $C[PL[i].EC]$
9: **end if**
10: **end for**
11: **end for**
12: **while** $S \neq \emptyset$ **do**
13: find service $e \in S$ with the largest $|C[e]|$
14: add e to SEC
15: remove $C[e]$ from S
16: **for all** service $s \in S$ **do**
17: remove $C[e] \cap C[s]$ from $C[s]$
18: **end for**
19: **end while**

Suppose n is the number of services in a BP and p is the number of probes. The diagnosis procedure is as follows.

1. Identify *faulty services*. Two kinds of faulty nodes can be found in this step.

   (a) Service that is the only source of a probe reporting an exception;

   (b) EC, whose input probe readings are all OK, but output probe reading flags an exception.

   For these obviously faulty services, we set all probes that cover the faulty services as handled. In this step, to find faulty services, the worst case is to scan the whole DM once, with a complexity of $O(pn)$. The complexity of finding and setting related probes is also $O(pn)$. So the total complexity of this step is $O(pn)$.

2. Produce *potential faulty sets*. For each remaining probe with exception, all its sources are saved in a potential faulty service set. These potential faulty sets will be used in Step 6. The step needs to scan the entire DM, with a complexity of $O(pn)$.

3. Identify *OK services*. If a probe has no exception, set all its source services that do not have a diagnosis result yet as OK services. The complexity of finding OK services is $O(pn)$.

4. Identify more *faulty services*. After some services are identified as OK services in Step 3, certain services become the only source for some unhandled exception probes. These services are now marked as faulty. We also set the probes of the faulty services as handled (same as Step 1). The complexity of this step is the same as Step 1, $O(pn)$.

5. Find *maybe services*. For remaining unhandled exception probes, if there is no faulty service clearly identified, their remaining source services are set as

maybe services. The complexity of this step is also $O(pn)$.

6. Re-introduce *maybe services* from potential faulty sets. If all services in a potential faulty set (from Step 2) are diagnosed as OK services, set all services in this set as maybe services.

   This problem is caused by the uncertainty of QoS violation propagation. Our goal is to find as many faulty services as possible. If there is a conflict about the diagnosis status of service, we would prefer to report the service as a faulty or maybe service. The complexity of this step is also $O(pn)$.

7. Identify *shielded services*. The remaining services are shielded services, i.e. certain faulty service may shield them and their status cannot be known for certain. Based on the assumption that most services are OK services, we treat all shielded services as OK services in the following implementation.

The overall complexity of the diagnosis algorithm is $O(pn)$.

We next show a diagnosis example. In Fig. 2, suppose only one input probe can be installed for every service, probe A is the input probe for service BulkMailQuoteB (union of probes 2,3) and probe B is the input probe of service SelectVendors (union of probes 11-15).The sources of probe A are service AddressQuoteA, AddressQuoteB, and the sources for probe B are PrinterQouteA, PrinterQuoteB, PrinterQuoteC, BulkMailQuoteA, and BulkMailQuoteB. Assume services AddressQuoteA and BulkMailQuoteB have QoS violations and probes A, B, and 4 have detected QoS problems. The diagnosis algorithm works as follow.

1. Step 1 finds faulty service BulkMailQuoteB by probe 4.

2. Step 2 produces potential faulty sets: {AddressQuoteA, AddressQuoteB} for probe A, {BulkMailQuoteA, BulkMailQuoteB, PrinterQuoteA, PrinterQuoteB, PrinterQuoteC} for probe B.

3. Step 3 identifies OK services: service SelectVendors from OK probe 16.

4. In Step 4, no extra faulty service is identified.

5. Step 5 finds maybe service AddressQuoteA and AddressQuoteB.

6. Step 6 introduces no maybe service.

7. Step 7 identifies 4 shielded services: BulkMailQuoteA, PrinterQuoteA, PrinterQuoteB, and PrinterQuoteC.

4. Predicates on Probes

In the diagnosis algorithm presented above, every probe is used to monitor the end-to-end response time from the beginning of a business process to its current location. Every ancestor service leading to a probe could cause an end-to-end response time delay. But some faults may become non-observable if other services have shorter than expected execution time that causes the delay to be compensated before a probe.

To have a more precise knowledge about individual services, we also compare response time data between probes. Since the temporal intervals between probes are smaller than the temporal intervals from the beginning of the BP to probes, we can have more precise fault detections.

4.1. PoP Definition

To fully utilize the information collected from selected probes, we define *predicate-on-probes* (PoP) to check the service(s) between two probes. There are two types of PoPs in a system: *inter*-PoP and *intra*-PoP. *Inter*-PoP is a PoP defined based on two probes on different services. Usually, an *inter*-PoP is between a service's output probe and the next (on the BP) EC's input probe. Fig. 3(a) is an example of *inter*-PoP, between the output probe of service 0 and the input probe of service 2. *Intra*-PoP is defined on a single service's input and output probes. By comparing an input's arriving time and the corresponding finish time, *intra*-PoP can decide if a service is faulty. In Fig. 3(b), an *intra*-PoP is defined for service 2.

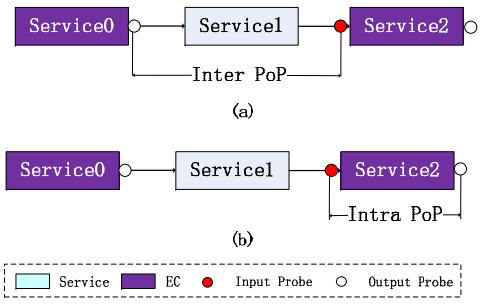

Figure 3. PoP Examples

4.2. PoP Generation

A PoP is defined by four tuples, p =<*beginEC*, *endEC*, *type*, *path*>. *beginEC* is the service ID of the beginning probe. *endEC* is the service ID of the end probe. Type shows it is either *inter*- or *intra*-PoP; for *intra*-PoP, *beginEC* and *endEC* are the same. Path, which is only for *inter*-PoP, identifies the parent service ID on the input path.

The PoP generation algorithm is shown in Algorithm 3. In Algorithm 3, *intra*-PoP and *inter*-PoP are generated for every EC. For *intra*-PoP, the service itself is a cause. For an *inter*-PoP $inter$, suppose $bProbe$ is its $beginEC$'s output probe and

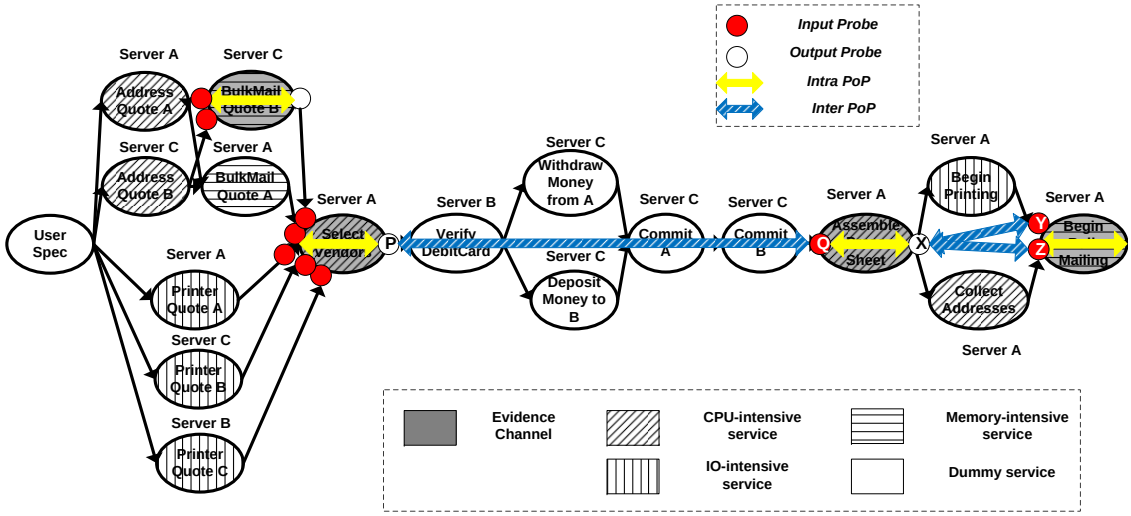

Figure 4. Probes and PoPs in PrintAndMail

$eProbe$ is the corresponding input probe of its $endEC$, the potential source set of $inter$ is the difference of $eProbe$ and $bProbe$'s source set. Suppose the number of EC is k, then the number of *intra*-PoP will also be k, the number of *inter*-PoP is $O(k^2)$. The time complexity of generating a PoP is $O(n)$, so Algorithm 2 is $O(k^2 n)$.

In Fig. 4, three *inter*-PoP's can be defined. One is between probes P and Q, with the sources of VerifyDebitCard, WithdrawMoneyfromA, DepositMoneytoB, CommitA and CommitB. The second one is between probes X and Y, with the source of BeginPrinting. The last one is between probes X and Z, with the source of CollectAddresses. If only one input probe can be detected for BeginBulkMailing, then the second *inter* PoP and the third *inter* PoP will be combined into one PoP, which covers both BeginPrinting and CollectAddresses. Four PoPs for services BulkMailQuoteB, SelectVendors, AssemblePressSheet and BeginBulkMailing are generated for the PrintAndMail Scenario.

Algorithm 3 PoP Generation Algorithm

Input: Dependency matrix DM; Probe list PL; Evidence channel set EC; Service list N.
Output: A New DM.

initial $PoPID = PL.size$
for all $e \in EC$ **do**
  set PoPList[PoPID]=<e,e,"intra",null>
  $\forall i, DM[PoPID][i] = 0$
  set $DM[PoPID][e.id] = 1$, $PoPID = PoPID + 1$
  for all $p \in e.parent$ **do**
    find the nearest EC e' to p
    set PoPList[PoPID]=<e',e,"inter",p>
    $\forall i, DM[PoPID][i] = 0$
    for all $s \in \neg(e' \bigcup e'.pred) \bigcap (p \bigcup p.pred)$ **do**
      set $DM[PoPID][s.id] = 1$
    end for
    set $PoPID = PoPID + 1$
  end for
end for

5. System Support for Dependency Matrix-based Diagnosis

We have implemented the diagnosis support in Llama. Figure 5 shows an example BP running on Llama. In the example, all services are deployed on the Accountability Service Bus (ASB), which is our version of the Enterprise Service Bus (ESB). In addition, Accountability Authority (AA) and Agents are the main Llama components. AA is responsible for the deployment and configuration of the accountability framework as well as performing fault diagnosis. Agents are selected by AA to collect data from Llama ASB and to decide the runtime status of probes and PoPs. For inter-PoPs, an Agent needs to communicate with another Agent to inspect its run-time status. For example, in Figure 5, Agent B pulls data from Agent A to decide the runtime status of the inter-PoP between the two evidence channels. Agents send all abnormal runtime data to AA. In this way, AA and Agents collaborate to perform run-time process monitoring and QoS fault diagnosis.

5.1. Llama Diagnosis Engine

Figure 6 shows the detailed architecture of Llama components, including AA, Agents and ASB. AA is comprised of three main modules:

1. **Management Gateway** is the portal to allow a user to configure BP structure with its QoS constraints.

2. **Agent Deployment** is in charge of Agent selection and initiation.

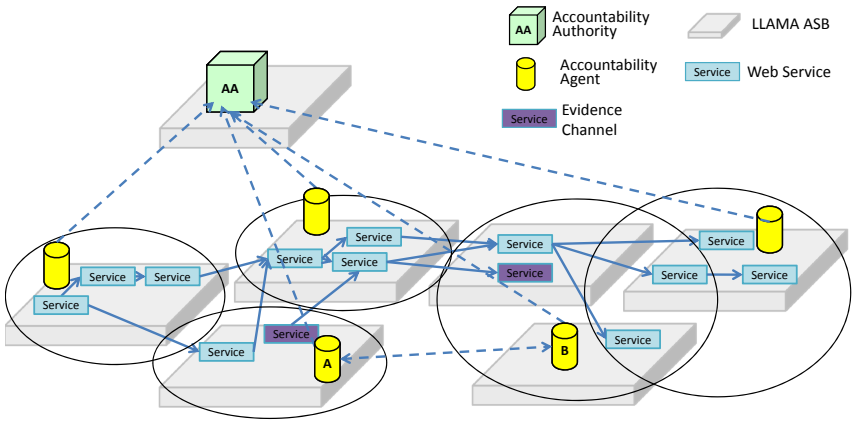

Figure 5. QoS Accountability Framework

3. **Diagnosis Engine** performs evidence channel selection and dependency matrix generation before runtime. During runtime, when Exception Receiver receives exceptions from agents, the diagnosis engine will investigate faulty sources.

Agents are the intermediaries between ASB and AA. They are responsible for the following tasks: (1) configuring evidence channels on ASB, (2) communicating with other Agents to perform EC data analysis, (3) reporting exceptions to AA, and (4) initiating error origin investigation upon the request of AA. An Agent has the following modules:

1. **ASB Configurator** receives instructions from AA and requests ASB to set up bindings between Agents and evidence channels.
2. **EC Data Receiver and Analyzer** receives timestamps of evidence channels pushed from ASB and stores them in the Monitoring Data Repo. For input timestamp, the Analyzer specifies the status of an input Probe and inter-PoPs whose endEC is the current service. For output timestamp, the analyzer checks the status of an output Probe and the intra-PoP of current evidence channel.
3. **Exception Reporter** reports all Probes or PoPs whose behavior is abnormal to AA.
4. **EC Data Retriever** receives instructions from AA to verify those maybe faulty services. It pulls information from the ASB log for services that are not selected as evidence channels.
5. **Error Origin Investigator** requests further information from ASB to determine if the source of an error is due to network, host, or the service itself once a problematic service is identified by AA.

Llama ASB provides an API and framework for Agents to collect service performance data. Data can be pushed or pulled, and collected and sent at configurable intervals. There are three main modules in ASB:

1. **Configuration Gateway** provides the functionality of configuring data dispatcher frequency and bindings between Agents and evidence channels;
2. **Profiling Interceptors** intercept service request and response message to take both begin and end timestamps of service invocation. These timestamps are stored in the ASB log;
3. **Profiling Data Dispatcher** sends runtime data of evidence channels to Agents with a pre-defined frequency.

5.2. Diagnosis Flow

The DM-based diagnosis is conducted as follows. Before runtime, EC selector & DM generator on AA first selects evidence channels and generates the corresponding Dependency Matrix with process configuration information from Management Gateway. With the evidence channel selection result, Agent Informer selects Agents from the Agent Repo and informs ASB Configurator on Agent selections. In the meantime, Agent Informer needs to update the evidence channel information to EC Data Repo on Agent for the future use of Agent communication. Once Agents build the bindings on ASB through Configuration Gateway, the preparation job is done.

At runtime, Profiling Interceptor records input and output timestamps during service executions. Profiling Data Dispatcher on Llama ASB sends the timestamps to their corresponding Agents with a pre-defiened frequency (in our experiment, it is 1 per 2sec, 1 per 5sec and 1 per 8sec). When EC Data Receiver receives runtime data, it will store the data in the Monitoring Data Repo. Meanwhile, EC Data Analyzer will analyze the status of corresponding Probes and PoPs. To decide the status of an inter-PoP, Agent of the endEC needs to pull information from the Agent of the beginEC and calculate the time difference. With the time difference, Agent will compare it with the time threshold stored in the EC Data Repo and decide the status of the corresponding probe

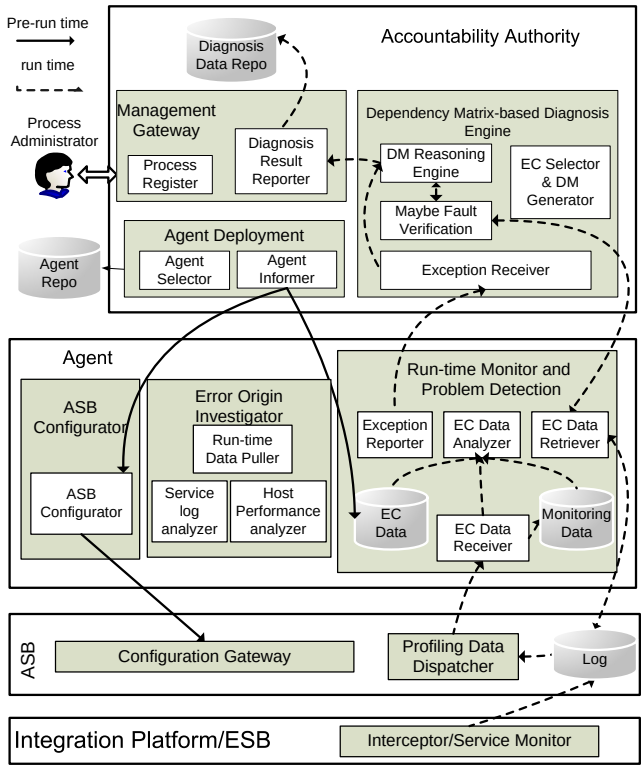

Figure 6. Llama Accountability architecture with DM diagnosis

or PoP. Exception Reporter on Agent sends all abnormal Probes or PoPs to AA. Once AA receives exceptions, DM Reasoning Engine will be triggered. The actual status of maybe services will be verified through Agents. Error origin of identified faulty services will be further investigated. The final diagnosis result is stored in Diagnosis Data Repo. Faulty services identified will be replaced by other services with the same or similar functionalities [12].

6. Empirical Results

6.1. Experiment Settings

The PrintAndMail BP is implemented in our lab and used to test the monitoring and diagnosis performance of the LLAMA middleware. To simulate real-world service behaviors, we have implemented four types of services: i.e. CPU-intensive, IO-intensive, memory-intensive, and simple services. As shown in Figure 4, there are five CPU-intensive services, four IO-intensive services, three memory-intensive services and five simple services.

The 17 services are deployed on three servers running Debian Linux OS, including two IBM servers, labelled as A and B, (with quad 3.06GHz CPU and 3.5GB memory) and one HP server C (with quad 2.8GHz CPU and 6GB memory). An agent is deployed on server A and another on server C. AA is on server C. Apache ODE [13] BPEL engine is deployed on server C. As shown in Figure 4, four evidence channels are selected in the process so that data from these services are being continuously pushed to the agents.

To profile its execution time, each service is invoked 100 times individually before BP execution. During profiling, we record each service response time on three system locations: the server side, the ASB side and the client side. The average response time recorded on the ASB and the service side has a difference less than 1 ms. However, the average response time recorded from the client side has about additional 80 ms due to network delay.

In our experiment, we set the fault threshold for each service response time by the sum of its *average response time* $+3*$ *standard deviation* $+$ *network delay*$+$ *BPEL engine delay*. If a service's response time exceeds that threshold, the service is considered faulty.

6.2. Monitoring Overhead

We set up five monitoring scenarios to measure system monitoring overhead:

1. execute with all interceptors turned off, ensuring that there is no data collection overhead,
2. execute with all profiling interceptors activated, i.e. all data are collected by local middleware,
3. the same as scenario 2, except that 4 evidence channels push data at 8 seconds intervals to agents,
4. the same as scenario 3, except the data push interval is 5 seconds,
5. the same as scenario 3, except data push interval is 2 seconds.

We have measured that the average end-to-end process response time (from 100 service process executions) is about 12,500ms. The monitoring oveheads for all of the five scenarios from two separate test rounds are shown in Table 3. From Table 3, we can see that the overhead of data collection at ASB is less than 100ms, which is small compared to the cost of BPEL engine and network communication. Pushing data to agents creates an extra delay of 50 ms to 300ms on service process execution. Overall, the total delay due to data collection on ASB and triggering data push to agents is less 400ms, which is about 3% of the average process response time.

6.3. Diagnosis Performance

We now present the diagnosis performance, including diagnosis correctness, identification and investigation overhead, and diagnosis response delay. Diagnosis response delay is the time difference between process finish time and diagnosis finish time.

Table 3. Monitoring Overhead (ms)

	data collection on ASB	ASB + data push @8s	ASB + data push @5s	ASB + data push @2s
Round 1	74ms	247ms	112ms	190ms
Round 2	84ms	232ms	313ms	363ms

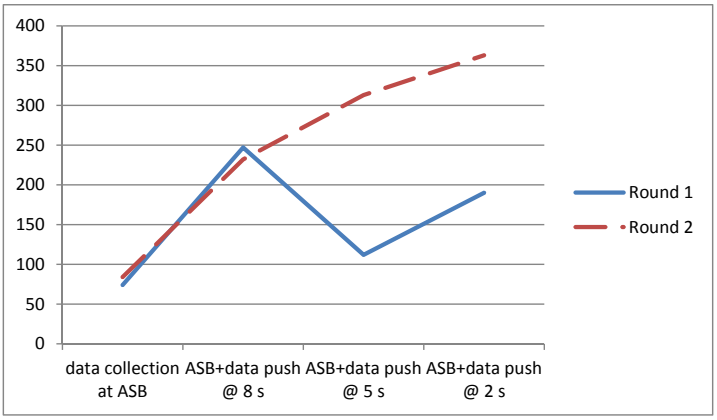

Figure 7. Monitoring Overheads(unit:ms)

The *diagnosis completeness ratio* shows the percentage of faults correctly identified by AA. For example, if 2 out of 4 faults are identified, the completeness ratio is 50%. The *false positive ratio* reflects the percentage of incorrect reports for non-faulty services. For example, if 1 out of 10 non-faulty services is diagnosed as faulty, the false positive ratio is 10%. Identification and investigation overhead is comprised by two parts, identification time and investigation time.

Identification overhead is the time used by the diagnosis process, which includes dependency matrix-based diagnosis time and verification time for maybe services (if applicable). Investigation overhead is the time used by specifying error origin for faulty services.

Three strategies are used to report *maybe* services: (A) report all *maybe* services as *faulty*; (B) report all *maybe* services as *normal* services; (C) verify the actual status of *maybe* services by pulling information from ASB through Agents.

In the experiment, we inject different types of delays (1 second, 3 seconds or 5 seconds) in randomly selected 1 or 3 services in the business process. For each scenario, the test was run 25 times. Tables 4 and 5 show the diagnosis performance, including average *completeness ratio*, *false positive ratio*, *Identification time*, and *Investigation time*, as well as the average diagnosis response delay. Each entry has three values, corresponding to reporting *maybe* services as *faulty*, *normal* or by actual identification.

In Tables 4 and 5, since verifying the actual status of *maybe* services only reduces the false positive ratio, but does not affect the completeness ratio, the completeness ratios of Strategies A and C are the same. They increase with the injected delay duration. Since we assign a large threshold to each service with respect to its average response time, when the injected error is only 1 second delay, it cannot be detected by the system and the completeness ratio is 0. When the delay duration increases from 3 seconds to 5 seconds, the completeness ratio increases from about 50% to about 80% (Figures 8) for one-error case and from about 60% to 80% (Figures 9) for three-error case . Obviously, the increasing completeness ratio is due to the fact that the longer the delay is, the easier it is to catch it. However, some 5 second delays cannot be caught. For example, in Figure 4, the average response time for PrinterQuoteB is only 270.68ms. However, the time threshold for AddressQuoteA and BulkMailQuoteB is 7800ms. Even when a 5-second delay is injected in PrinterQuoteB, it still may not be detected.

False positive ratio for strategy A grows with error number and delay duration. In our experiment, average false positive ratio distributes between 3.5% and 15.5%. The false positive problem is mainly introduced by the strategy that we report all *maybe* faults as identified faults. For strategy B, we can see that if we treat all *maybe* services as non-faulty, the false positive ratio drops to 0. In fact, the actual faulty service out of *maybe* services is always less than 50%. In the actual system,

Table 4. Diagnosis Performance for 1 Injected Error (maybe service = faulty, normal, actual)

	Complete_Ratio	FaultPos_Ratio	IdentificationTime	InvestigateTime	Diag_RSP
1 sec delay	0, 0, 0	0, 0, 0	0, 0, 0(ms)	0, 0, 0(ms)	0s
3 sec delay	50%, 25%, 50%	3.5%, 0, 0	2.7, 2.7, 46.1(ms)	56.1, 27.3, 29.1(ms)	2.4s
5 sec delay	81.3%, 18.8%, 81.3%	9.0%, 0, 0	1.8, 1.8, 110.5(ms)	126.1, 11.1, 41.8(ms)	2s

Table 5. Diagnosis Performance for 3 Injected Errors (maybe service = faulty, normal, actual)

	Complete_Ratio	FaultPos_Ratio	IdentificationTime	InvestigateTime	Diag_RSP
1 sec delay	0, 0, 0	0, 0, 0	0, 0, 0 ms	0, 0, 0 ms	0s
3 sec delay	58.1%, 37.2%, 58.1%	4.5%, 0, 0	3.1, 3.1, 48.9 ms	130.6, 60.7, 98.5 ms	2.1s
5 sec delay	79.4%, 40.6%, 79.4%	15.5%, 0, 0	3.3, 3.3, 115.6 ms	240.6, 67.2, 132.1 ms	1.7s

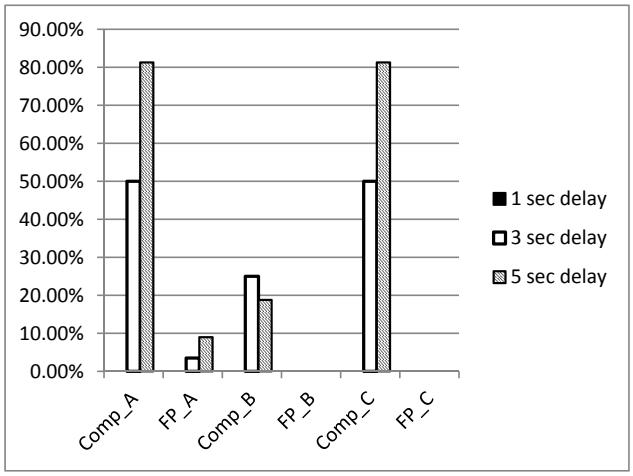

Figure 8. BP Completion Ratio and False Positive Ratio (one error)

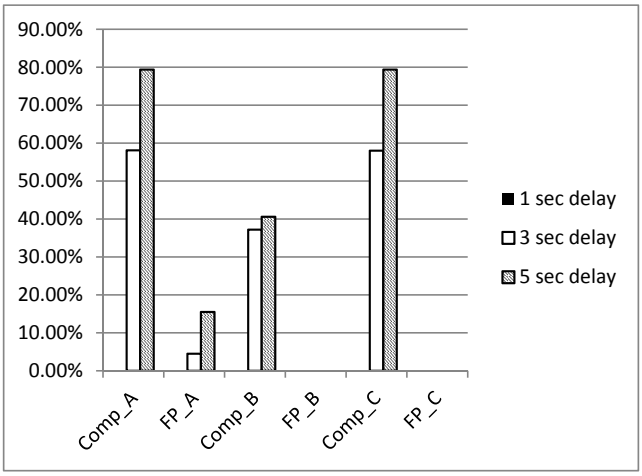

Figure 9. BP Completion Ratio and False Positive Ratio (three errors)

we can double check those *maybe* services before a system reconfiguration.

Different from other NP-hard diagnosis algorithms, the dependency matrix based diagnosis has a polynomial time complexity. As shown in Tables 4 and 5, the identification time for Strategies A and B is short, less than 4ms, which is negligible. For Strategy C, network delay dominates the verification time. Verification part can cause an extra 100 ms delay. The identification time for all scenarios is less than 120ms.

Error origin investigation is an optional feature in the system. For each identified error, its error origin can be detected through investigation. Similar as *maybe* service verification, network delay dominates the investigation time. The investigation time for different scenarios distributes in the range of 40ms to 250ms. Since Strategy A suffers from high false positive ratio, the number of services need to be investigated is greater than the other two strategies, Strategy A has the highest investigation time. On the other hand, Strategies B and C have the same false positive ratio, but Strategy B has a lower completeness ratio, so the number of services to be investigated for Strategy B is less than that of Strategy C and it has a shorter investigation time.

Since the diagnosis time is relatively short, the diagnosis response delay (Diag_RSP) is mainly caused by the data push delay on ASB. For the experiment reported in Table 4, the data push interval is set to 5s. We can see that the average diagnosis response delay is around 2s, which is about half of the data push interval of 5s.

Although diagnosis sensitivity still depends on the setting of threshold and service location in the process, due to the usage of PoP, the diagnosis sensitivity of dependency matrix based diagnosis is much better than those diagnosis approaches only consider end-to-end information. For example, a delay on VerifyDebitCard may be covered by its good-behavior predecessors and cannot be caught by end-to-end time. However, it can be detected by the inter POP between SelectVendors and AssemblePressSheet in our diagnosis.

6.4. Summary

From the experiment result, we can see that the dependency matrix based diagnosis is efficient for service monitoring and accountability management. Diagnosis overhead of dependency matrix based diagnosis is very small, and its diagnosis performance, in terms of completeness ratio, false positive ratio and diagnosis sensitivity, is reasonably good. Many diagnosis system attributes, such as diagnosis response speed and diagnosis sensitivity can be specified by the user by setting the data dispatching frequency and service time threshold.

7. Related work

7.1. Dependency Matrix Based Diagnosis Projects

Among different approaches designed for multiple fault diagnosis, dependency matrix based method is known to be computationally efficient and also easier to apply in real world. It has been successfully utilized in a variety of systems [18, 19]. However, for multiple fault probabilistic diagnosis, its result may not be as good as some rigorous but computationally intractable (NP-hard) diagnosis approaches (such as Bayesian Network). In this paper, we apply the dependency matrix method to QoS diagnosis in SOA. To fully use the monitoring information, we introduce PoP, which can acquire more detailed information by comparing neighboring probes.

Rish et al. [17, 18] apply dependency matrix-based diagnosis in network management. Different from network management, in SOA, every service's QoS is influenced by its predecessor services, so the potential cause of each probe is uncertain. To handle this uncertainty, we modify the original dependency matrix diagnosis to fit it in SOA as discussed in Sec. 3.3.

Previous works [14, 18] have also studied the probe selection problem for the purpose of network management. Ozmutlu et al. [14] propose to optimally select a subset of ping-like probes to monitor networks by using Zone Recovery Methodology (ZMR). ZMR works for unknown topology structure. But in SOA the process structure usually is available, so ZMR is not suitable for SOA. For active probing [18], Rish et al. pre-select the minimal set, which covers all nodes, to monitor at run-time. After error is detected, the most-informative next probe must be activated. During diagnosis, active probing requires pulling information from probes dynamically. In our study for SOA systems, we collect information only from pre-selected probes and conduct diagnosis based on reported data, without creating new probes. One reason is that process instances are not produced as often as network packets. So creating new probes may not collect too much new information to improve the diagnosis correctness. In [18], the conditional probability of each node is required for probe selection. In our method, only PDM is needed and a set-covering algorithm is used to make probe selection more efficient and practical.

Agarwal et al. [1] apply dependency graphs and run-time behavior models to design algorithms for rapid root cause identification in case of problems occur in e-business. The concept of dependency graph is similar to our dependency matrix. However, they only use the dependencies to update the service servility value but not to reason the root cause directly. The primary contribution of this paper is that response time threshold is dynamically constructed for each component by observing its behavior. This solves the problem that it is very difficult and error prone for the system administrator to configure a threshold for a component without extensive benchmarking experience. Meanwhile, their work has several limitations compared to our work:

(1) their approach is based on single-fault assumption; but ours can solve multiple-fault diagnosis; (2) their approach only considers end-to-end time, but never makes comparison among probes. Our system involves both *inter*-PoP and *intra*-PoP to fully use the detected information.

7.2. QoS Diagnosis Projects

Other diagnosis approaches, such as Bayesian network and model-based diagnosis, are also applicable in SOA. However, there are two issues on using Bayesian network in SOA systems: (1) there may be insufficient historical knowledge to train a workable Bayesian network; (2) the complexity of Bayesian network diagnosis is NP-Hard. Zhang et al. propose a QoS-aware diagnosis framework in SOA [23] using Bayesian network diagnosis. Compared to Bayesian network diagnosis, dependency matrix-based diagnosis is more efficient at run time and does not need any expert knowledge about services. In the previous version of the Llama middleware [11], a Bayesian diagnosis engine was used. Since only end-to-end timestamps were considered in the diagnosis, agents do not need to communicate with each other. In the current Llama version, with both inter- and intra-PoPs, the dependency matrix-based diagnosis achieves a better diagnosis sensitivity without bringing much system overhead.

Lee et al. [10] provide a common architecture for the distributed diagnosis of Internet faults using autonomous agents. CAPRI diagnoses faults using probabilistic relational models (PRMs) to combine the strengths of probabilistic Bayesian inference with the descriptive power of first-order logic. This approach is quite similar to our Bayesian network diagnosis [23] where every service could be a deterministic or chance node. So it also suffers from the high time complexity of Bayesian network diagnosis and lower diagnosis sensitivity due to lack of comparison among different evidence channels.

Wang et al. [20] integrate monitoring and diagnostic services into the QoS management framework. A graphical model-based approach, i.e. causal networks, is used in root cause diagnosis. However, causal network reasoning request system knowledge, which may be not available. Without the capacity to deal with uncertain and incomplete information, the causal network reasoning may be not an ideal model for QoS diagnosis in SOA. Moreover, in [20], even error origin, which is investigated from ASB in our system, is reasoned with root causes, this strategy will significantly increase the size of their graph, and will cause long diagnosis overhead.

Ardissono et al. [2] propose a consistency-based diagnosis approach that spans across individual services to enhance fault analysis in SOA. In their framework, each web service has a local diagnosor. The global diagnoser coordinates the local diagnosers. Their system architecture, which has a global diagnoser (similar to AA) and multiple local diagnosers (similar to Agents), is similar to our Llama project. However, the consistency-based diagnosis approach presented requires multiple message exchange steps between a global diagnoser and a local diagnoser before the root cause is ruled out, while our diagnosis approach can reason out the root causes with a one-time message exchanged between the AA and the Agents (except for maybe service verification part, which is optional). Furthermore, the diagnostic correctness of their approach relies on the ability to collect data from every service. Our accountability framework requires evidence collection at a subset of locations and therefore reduces the monitoring cost.

WS-Diamond [5] also aims to provide a framework with self-diagnosis and self-repair capabilities for Web services. In the WS-Diamond project, model-based diagnosis is used for functional and non-functional diagnosis. Unlike their approach, our work does not need historical knowledge about services, and is more appropriate for SOA systems that do not have a static process structure.

7.3. Commercial Products

Some commercial middleware, such as CapeClear [4] and RTI [16], are available on the market. Both CapeClear and RTI products have the functionality of monitoring service performance inside a business process. However, unlike the Llama middleware, CapeClear BAM is in lack of the capability of automatic diagnosis and recovery. It usually reports information via a dashboard or email alerts to human, who has to manually initiate diagnosis and corrective actions. RTI product has very powerful monitoring, diagnosis and error investigation functions. However, it is designed more on identifying problems during design and initial integration. The Llama middleware is more concerned with runtime diagnosis and recovery to provide management of flexibility in SOA.

8. Conclusion

With the popularity of SOA-based applications, there is an increasing need for managing business process QoS at runtime. To address this need, we present a framework for runtime service behavior monitoring, service fault diagnosis, and service process recovery [12]. The framework uses a dependency matrix to denote the relationship between performance probes and services monitored by each probe. A diagnosis algorithm based on the dependency matrix and the probe data is then used to identify potential faulty services.

We have implemented the diagnosis algorithms and runtime support in the Llama middleware. We have also tested the diagnosis performance using realistic services deployed on networked servers. The experiment study shows that the diagnosis mechanism performs well in our test environment. We plan to deploy the Llama middleware in large-scale network to study its practicality.

References

[1] M. K. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, and A. Sailer. Problem determination using dependency graphs

and run-time behavior models. In *Proceedings of the 15th IFIP/IEEE International Workshop on Distributed Systems*, pages 171–182, 2004.

[2] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, and M. Segnan. Enhancing web services with diagnostic capabilities. *Third IEEE European Conference on Web Services (ECOWS 2005)*, Nov 2005.

[3] M. Bichler and K. Lin. Service-oriented computing. *IEEE Computer*, 39(3):99–101, March 2006.

[4] CapeClear. Capeclear BAM. *http://developer.capeclear.com/*, 2008.

[5] L. Console and M. Fugini. WS-DIAMOND: an approach to web services, diagnosability, monitoring and diagnosis. In *Proceedings of the International e-Challenges Conference*, Oct 2007.

[6] G. Cooper. The computational complexity of probabilistic inference using bayesian belief networks. *Artificial Intelligence*, 42(2-3):393–405, 1990.

[7] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[8] J. D. Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[9] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the 4th international Symposium on Integrated Network Management IV*, pages 266–277. Chapman Hall, 1995.

[10] G. Lee. CAPRI: a common architecture for autonomous, distributed diagnosis of internet faults using probabilistic relational models. In *the First Workshop on Hot Topics in Autonomic Computing (HotAC I) in conjunction with the 3rd IEEE International Conference on Autonomic Computing (ICAC-06)*, 2006.

[11] K. Lin, M. Panahi, Y. Zhang, S.-H. Chang, and J. Zhang. Building accountability middleware to support dependable SOA. *IEEE Internet Computing*, 13:16–25, March 2009.

[12] K. Lin, J. Zhang, Y. Zhai, and B. Xu. The design and implementation of service process reconfiguration with end-to-end QoS constraints in SOA. *Service Oriented Computing and Applications*, 4:157–168, 2010.

[13] ODESource. Ode 1.2. *http://ode.apache.org/user-guide.html*, 2008.

[14] H. Ozmutlu, N. Gautam, and R. Barton. Zone recovery methodology for probe-subset selection in end-to-end network monitoring. In *Network Operations and Management Symposium*, pages 451–464, 2002.

[15] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented computing: State of the art and research challenges. *IEEE Computer*, 40:38–45, November 2007.

[16] Real-Time Innovations. RTI monitor. *http://www.rti.com/docs/RTI_Monitor.pdf*, 2010.

[17] I. Rish, M. Brodie, and S. Ma. Intelligent probing: A cost-efficient approach to fault diagnosis in computer networks. *IBM Systems Journal*, 41(3):372–385, 2002.

[18] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez. Adaptive diagnosis in distributed systems. *IEEE Transactions on Neural Networks*, 16(5):1088–1109, 2005.

[19] F. Tu and K. R. Pattipati. Rollout strategies for sequential fault diagnosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(1):86–99, 2003.

[20] G. Wang, C. Wang, A. Chen, H. Wang, C. Fung, S. Uczekaj, Y.-L. Chen, W. Guthmiller, and J. Lee. Service level management using QoS monitoring, diagnostics, and adaptation for networked enterprise systems. *IEEE Enterprise Computing Conference (EDOC)*, pages 239–248, Sep 2005.

[21] J. Zhang, Y. Chang, and K. Lin. A dependency matrix based framework for qos diagnosis in soa. In *Proc. of IEEE International Conference on Service-Oriented Computing and Applications(SOCA)*, 2009.

[22] Y. Zhang and K. Lin. Efficient algorithms for selecting optimal data collection locations in business process management. In *IEEE International Conference on e-Business Engineering (ICEBE'08)*, October 2008.

[23] Y. Zhang, K. Lin, and J. Y. Hsu. Accountability monitoring and reasoning in service-oriented architectures. *Service Oriented Computing and Applications*, 1(1):35–50, 2007.