2009 Second International Conference on Dependability

Diagnosis Service for Software Component and its Application to a Heterogeneous Sensor Data Management System

Thi-Quynh Bui,
Oum-El-Kheir Aktouf, Michel Dang
LCIS Laboratory
Valence, France
{thi-quynh.bui, oum-el-kheir.aktouf,
michel.dang}@lcis.grenoble-inp.fr

Levent Gürgen
National Institute of Informatics
Tokyo, Japan
levent@nii.ac.jp

Claudia Roncancio
LIG Laboratory
Grenoble, France
claudia.roncancio@imag.fr

Abstract - **Component-based software development paradigm enables the construction of complex applications by assembling existing self-contained components. The cost of the software development process can then be sharply reduced. Current software systems are becoming more distributed and operate in highly dynamic environments. Therefore, dependability of component-based applications is an important research issue. Our contribution consists of a diagnosis service that enhances dependability of component-based applications. This diagnosis service provides the ability of detection and location of faulty components during runtime. In this paper, we describe the application of the proposed diagnosis service to an industrial case study that consists of a heterogeneous sensor data management system.**

Keywords – dependability; diagnosis; software component

I. INTRODUCTION

Component-based software development enables the construction of applications by assembling existing self-contained components with well defined interfaces. The cost of the software development process can then be sharply reduced. In addition, the use of replaceable software components simplifies the implementation and the maintenance of complex applications. Current software systems are becoming even more distributed and operating in highly dynamic environments. Thus, dependability of component-based applications is an important research issue.

In this context, most of the proposed approaches [1][2][3] are based on component replication and fault masking. Thus, results are guaranteed to be correct though some faults may corrupt the functioning of some application components. But such solutions are costly. Furthermore, software components are executed in complex environments which can evolve dynamically. Components can be installed or removed at any time and the application's structure can change at runtime. This constraint is difficult to manage with replication.

Our proposed approach is based on the system-level diagnosis technique that concerns the ability of fault-free components to determine the fault-state of the whole application. Advantages of our approach are application autonomy, cost-effectiveness and better usage of system resources. It adapts to dynamic evolutions of component-based applications.

System-level diagnosis is a process initially introduced for hardware systems by Preparata *et al.* [4]. It allows to determine the fault-state of distributed system components. Its basic idea is to use inter-component tests, *i.e.* a component tests another one to determine its fault state. System-level diagnosis may rely on centralized diagnosis or distributed diagnosis. In the former, it is assumed that a central reliable device monitors test execution and computes the system state on the basis of test results. In the latter, it is supposed that there is no central device and each fault-free component is aware of the whole system state. So, such algorithms are called self-diagnosis algorithms. Depending on the way tests are assigned, diagnosis algorithms can be static or adaptive. In static algorithms, test relations are established before the diagnosis process. On the contrary, they are dynamic in the adaptive strategy and depend on previous test results that indicate the state of some system components.

The proposed diagnosis service extends diagnosis concepts to software components and provides the ability of detection and location of faulty software components at runtime. Once the diagnosis process is completed, the system is able to isolate the diagnosed faulty components, by ignoring their output and initiating a reconfiguration operation such that the reliability of the system can be maintained in the long run.

This paper describes the application of our diagnosis service on an industrial case study. It illustrates our proposed solutions to the raised problems in the diagnosis process and their implementation in a concrete case study. The obtained results constitute premises for our goal of building fault tolerant applications in software component based systems. The detailed description of our diagnosis service is presented in previous papers [5][11].

This paper is organized as follows: Section II gives a global view of the proposed diagnosis approach. Section III describes the case study which is a heterogeneous sensor data management system. The diagnosis service's implementation in this case study is illustrated in Section IV. The obtained experimental results are given in Section V. Section VI gives some concluding remarks.

978-0-7695-3666-8/09 $25.00 © 2009 IEEE
DOI 10.1109/DEPEND.2009.28

II. DIAGNOSIS SERVICE

This section presents the proposed diagnosis service and cites the potentially raised problems when building such architecture.

A. *Global diagnosis service architecture*

The global architecture of the diagnosis service is described in Figure 1.

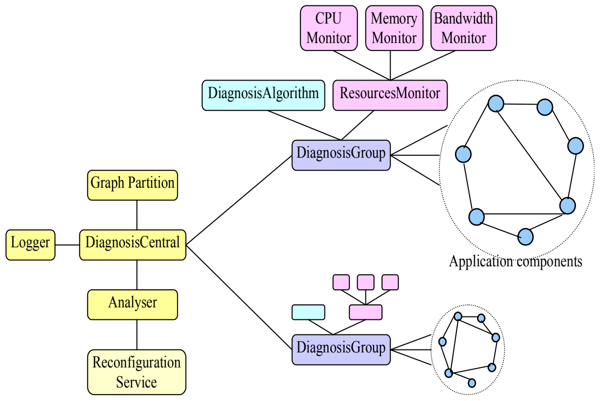


Figure 1. Diagnosis service architecture

Its main components are as following:

- *DiagnosisCentral*: is a control site, it has information on all diagnosable components in the system, for example: name, location, test cases, *etc*. It manages all diagnosis groups of the system and gets the global fault state of the diagnosis groups or the fault state of the whole system.
- *Graph Partition*: divides the system into diagnosis groups where inter-component tests are performed following a given test assignment. The partitioning approaches perform better than non-partitioning ones that cover the whole system by reducing unnecessary extra message traffic and processing time [7].
- *Logger*: takes the responsibility of error logging, for statistic analysis, system report, *etc*. This logger is useful for the purpose of group partitioning and the choice of an appropriate diagnosis algorithm. For example, we can group the frequently faulty components in a group and use an appropriate diagnosis algorithm more efficiently. This statistic information can also be used to determine the appropriate diagnosis period or diagnosis time.
- *Analyzer*: receives information on the faulty components. This information, such as error logging, test cases, *etc.*, will be analyzed for determining the type of the fault produced by the component and triggering the reconfiguration process.
- *Reconfiguration Service*: provides different strategies of system reconfiguration, for example, online repair, component replacement or system shutdown, *etc.* [9].
- *Diagnosis Algorithm*: provides different diagnosis algorithms, that may be selected by the application.
- *Resources Monitor*: observes the usage of resources of the system (CPU, memory, bandwidth, *etc.*) in order to provide the diagnosis process with available system resources thus avoiding to disturb the application.
- *DiagnosisGroup*: manages and schedules the inter-component tests according to the used diagnosis algorithm and the available resources.
- *Application Components*: are components of the application providing testing abilities.

B. *Potentially raised problems*

As our architecture is based on online inter-component testing, in practice, we need to study the following problems due to runtime system execution:

1. Test implementation: how to test a component and what to test? A component is considered as a black-box with provided interfaces and required interfaces. We integrate a test interface into components to provide testing facilities.
2. Concurrency: tests can interfere with the business functionality of the component. What should the component do if during the testing process it receives a regular business service request from other components?
3. Correctness of system states: on-line testing is carried out in the real environment. If the test manipulates and changes functional data of the component, it has to provide some roll-back mechanisms, ensuring the affected data during the test are set to the original state to keep the correctness of the system state.
4. Efficiency: Diagnosis approach has to be efficient, meaning the approach should well use the available resources to reduce the impact on system performances. In particular, the inter-component tests need to regard the system resources, *i.e.* memory, CPU time or bandwidth.

In the next section, we describe some possible solutions to tackle the problems listed above.

C. *Proposed solutions*

1) *Test implementation*

The principle of the proposed component testing approach is to integrate a test interface in the component. This interface provides testing facilities and the test cases of the component [6].

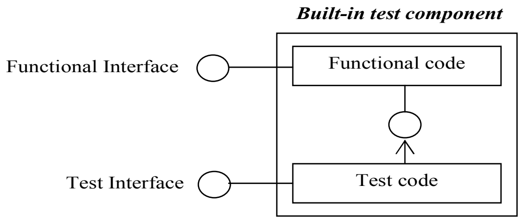


Figure 2. Built-in test component

A component contains a set of provided and required interfaces. Each provided interface is a set of operations that the component provides to other components, while each required interface is a set of operations that the component needs to perform its execution. In a similar manner, testing facilities are another service that the component provides to its environment. So the tester component can use this interface in the same way as the other functional interfaces (Figure 2).

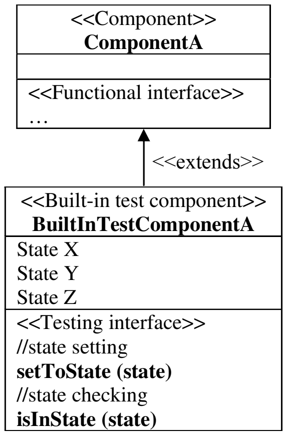


Figure 3. Concepts of built-in test component and testing interface

Generally, a component can be viewed as a state machine and requires state-transition testing.

Before the execution of a test, the tested component must be brought into the initial state required for a particular test. After test-case execution, the test must verify that the outcome (if generated) is as expected, and that the tested component resides in the expected final state. To reach this goal, the testing interface contains special purpose operations for setting and retrieving the internal state of a component, as shown in Figure 3.

A testing interface extends the normal functionality of the component and comprises operations for setting and getting internal state information. The state checking operation (`isInState(state)`) verifies whether the component is currently residing in a distinct logical state or not. The state setting operation (`setToState(state)`) sets the components' internal attributes to represent a distinct logical state.

More details on the testing interface and how to define test cases are described in [6]. Section IV below illustrates the implementation of built-in tests on the studied sensor data management system.

2) *Concurrency and correcteness of the systems*

The problem of concurrency occurs when the test and another component call the component function simultaneously. To ensure the correctness of the system, the business functionality should not be disturbed. Some solutions for both problems are described in the literature [8]:

- The component is blocked during the testing process. While the test is executed, the test and business requests received from other components are delayed until the end of the testing process. The system correctness problem can be resolved here by restoring the component to its original state after the testing process is finished.
- The component aborts the testing process. Every time another component requests the business service of the tested component, the testing process is aborted. Once aborted, the state of the component has to be restored.
- The component is cloned by the infrastructure before the test starts. The testing process is performed with the clone and the service process is performed with the original component. In this solution, the system correctness problem can be ignored. However, this option can be very resource consuming.
- The component test interface provides methods that allow test sessions. These methods ensure that test data and real data are not mixed during the testing process. In this solution, the problem of the correctness of the system state is resolved. This option puts an additional burden on the component developer.

It is worth noting that there is no optimal solution for figuring out this problem. Depending on the context, we will choose the most suitable solution. We will see in Section IV our choices in a concrete case study.

3) *Efficiency*

To improve the efficiency of the system, the scheduler component properly balances test execution and processing of the "normal" functionality. It employs test execution strategies to provide an appropriate trade-off between testing and core functionality in an intelligent manner. These strategies are listed below:

- Idle: the tests are executed if the tested component is idle.
- Resource threshold: the tests are executed if the resource availability exceeds a certain threshold for every individual resource.

In the next section, we will present an industrial case study - a sensor data management system. And then, apply our diagnosis service to this system to detect and locate faulty component.

III. SSTREAMWARE SYSTEM

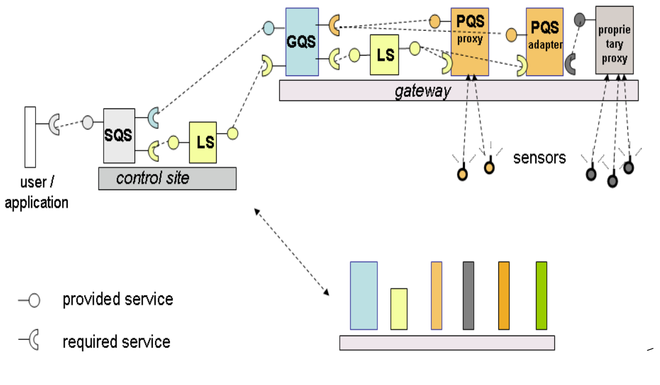


Figure 4. Architecture of the sensor data management system

SStreaMWare [12] is a service-oriented middleware for heterogeneous sensor data management. It is placed between applications and heterogeneous sensor farms. SStreaMWare provides a query service supporting a declarative query language to access data streams issued by heterogeneous sensors. It uses a service-oriented architecture to handle software heterogeneity and to operate upon a dynamic set of distributed sensors. It is implemented on the OSGi Service Platform [13].

SStreaMWare architectural entities consist of three querying services and one service discovery and binding service which are implemented as OSGi services deployed on distinct distributed platforms:

- Sensor Query Service (SQS),
- Gateway Query Service (GQS),
- Proxy Query Service (PQS),
- and Lookup Service (LS).

Figure 4 illustrates the service oriented architecture and the interaction between different services. Following sections give details on the implementation of these services.

A. *Sensor Query Service*

SQS is located at a control site, and used by users/applications. Its responsibility towards query evaluation concerns:

- accepting queries addressed to all the sensors of the environment managed by the control site ;
- decomposing queries and sending them to concerned gateways (which are localized thanks to the Lookup Service) ;
- possibly contributing to the query evaluation ;
- returning results as data streams.

A control site can be an OSGi gateway or another entity which can host the SQS (e.g., in a form of a web service in a web server). In this case, it is implemented as a remote service accessible by an IP address and a port number and hosted by an OSGi gateway. Used remote communication protocol is Java RMI (Remote Method Invocation). SQS is registered in a RMI registry which allows the clients to discover and invoke it.

B. *Gateway Query Service*

GQS is in charge of evaluating sub-queries concerning a given sub-region. At each gateway, there is one GQS providing a query service for the sensors managed by the proxy services of the gateway. The SQS sends the sub-queries to the relevant GQSs. The GQS re-decomposes the sub-queries in order to send them to concerned proxies/adapters which are localized thanks to the LS instance running on the gateway. After the sub-query evaluation, the results are then sent back by the GQS to the SQS.

C. *Proxy Query Service*

PQS is a query service which is related to one specific type of sensor. The implementation of this service is specific to sensors. Under a generic interface, PQS thus hides the heterogeneity of sensor software. PQSs are also catalogued by the LS, which enables GQSs to localize and use them. Typically, GQSs discover PQSs with lookup queries containing predicates on the type of the proxies; e.g., proxy of temperature sensors, proxy of a certain provider.

D. *Automatic Service Lookup and Binding Service*

The Lookup Service (LS) is a service discovery and binding service. This service manages a registry that contains information about different services present in the system.

IV. Diagnosis of the SStreaMWare system

In this case study, we analyze the diagnosis of the PQSs which are responsible of the connection between the sensor data management system and the sensors themselves. Note that each service in this case study is a component. Diagnosis groups are defined according to natural hierarchy of geographic locations. It means that a diagnosis group is a group of PQS components that are located in the same gateway.

Implementation of the diagnosis service within the SStreaMWare system needs the following components (Figure 5):

- DiagnosisCentral component.
- DiagnosisGroup component.
- Interfaces for the test and diagnosis in every component which should be diagnosed.

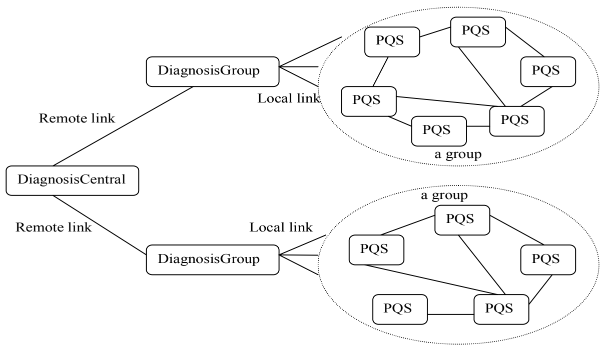


Figure 5. The connection between the component of diagnosis service and the system components

Connections between components in a group and the DiagnosisGroup component are local. Connections between the DiagnosisGroup and DiagnosisCentral components are remote using the Java RMI.

A. *DiagnosisCentral component*

DiagnosisCentral component is the central component which is responsible of:

- Collecting the useful component information for diagnosis: name, location of the component, test cases provided by components, *etc.*
- Partitioning the system into diagnosis groups.
- Managing all the diagnosis groups.
- Observing the state of all components in a group or in the whole system.

It provides a graphic interface that allows the user to choose diagnosis arguments, for example: criteria of the partitioning, number of diagnosis groups; diagnosis algorithm; diagnosis start: one time, periodic, diagnosis period; and resource threshold.

This set of information is defined for each diagnosis group and is transmitted to the corresponding DiagnosisGroup component.

B. DiagnosisGroup component

Each DiagnosisGroup component implements the *diagnosisService* interface and extends the *java.rmi.Remote* interface in order to allow the use of DiagnosisCentral remotely. The *diagnosisService* interface provides a service that diagnoses a component group according to the diagnosis parameters transferred by the DiagnosisCentral component.

This DiagnosisGroup component provides functions for:

- Collecting information on all components in its group.
- Supporting different diagnosis algorithms.
- Observing available resources in the system.
- Managing all inter-component tests in its group according to the diagnosis algorithm chosen by the user and considering the available system resources.

C. Integrated interfaces into PQSs components

Each PQS component provides two interfaces for the diagnosis, which are *diagnosisInterface* and *testInterface*.

- *diagnosisInterface*: allows to transmit information such as the name, the location and the provided test cases to DiagnosisGroup component. It also allows to receive test requests from DiagnosisGroup. It then analyzes received test requests to find out which components will be tested with which test cases.
- *testInterface*: is the testing interface of the component that contains all the test cases. The tester component uses this interface to test this component.

1) Test Interface

As mentioned in Section C of III, the PQSs provide a service which consists of the following principal functions:

- The interrogation method of sensors `evaluateQuery (QueryPlan qp, Receiver reciver).`
- The method `stopQuery(int QId)` that stops execution of the request with the given identifier.
- The method `getSensorsInfo()` return information about the sensors in its charge.
- The method `updateSensor (SID Integer, String attr, Object nValue, int priority)` to change or update information of its sensors.

After analyzing these components, we determined the corresponding state machine that is shown in Figure 6. There are two states: *active* and *idle*. The *idle* state represents a state where these components have no query on their sensors. On the contrary, the *active* state corresponds to the fact that components have queries on their sensors.

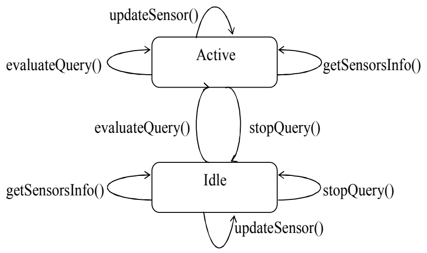


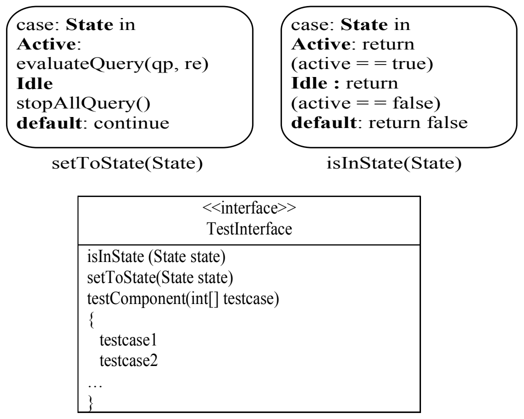
Figure 6. State machine of PQS components

From this state machine, test cases that correspond to any possible transaction are described in Table I.

TABLE I. THE TEST CASES OF PQS COMPONENTS

Test cases	Initial state	Input	Tested method	Final state	Outcome
1	idle	a query plan qp a receiver re	evaluateQuery(qp, re)	active	tuples (data generated by the sensor)
2	active	a query plan qp a receiver re	evaluateQuery(qp, re)	active	tuples (data generated by the sensor)
3	active	query identifier QId	stopQuery(QId)	idle	The query is stopped
4	idle	QId	stopQuery(QId)	idle	
5	active		getSensorsInfo()	active	The sensor information
6	idle		getSensorsInfo()	idle	The sensor information
7	active	Sid : sensor identifier Attr : name of the attribute nValue : new value priority	updateSensor(SId, attr, nValue, priority)	active	The updated sensor information
8	idle	Sid Attr nValue priority	updateSensor(SId, attr, nValue, priority)	idle	The updated sensor information

The PQS component comprises only two states *active* and *idle* representing respectively the presence or not of queries on its sensors. So, the implementation of both operations `isInState(state)` and `setToState(state)` is simple, as follows:



```
case: State in
Active:
evaluateQuery(qp, re)
Idle
stopAllQuery()
default: continue
```

setToState(State)

```
case: State in
Active: return
(active = = true)
Idle : return
(active = = false)
default: return false
```

isInState(State)

```
<<interface>>
TestInterface
isInState (State state)
setToState(State state)
testComponent(int[] testcase)
{
  testcase1
  testcase2
…
}
```

Figure 7. The testing interface

The testing interface (Figure 7) is an interface within a component that provides the testing functionality to allow the component own test. Besides both functions `setToState(state)` and `isInState(state)`, this testing interface also exports the function `boolean [] testComponent(int [] testcases)` so that other components can test this component. This function consists of the implementation of all test cases provided by this component. Table *testcases* is transmitted as a parameter containing the test cases that the tester component needs to test this component. This function returns a table of results corresponding to the executed test cases.

2) The concurrency and correctness problems

In the case of the heterogeneous data sensor management system, the data is sent periodically by sensors in real time. That is why the first solution of the concurrency and correctness problems (Section 2 of C of II) is to be avoided. The third solution consumes a lot of memory. So the second and the fourth options are probably the best suited ones.

- For test cases with the `evaluateQuery (queryPlan qp, receiver re)` and `stopQuery (QId)` methods: we opt for the fourth solution.
    - `evaluateQuery(queryPlan qp, receiver re):` we create our own query plan for the request and a receiver to receive data from sensors. So they do not interfere with the normal function of the component or the system.
    - `stopQuery(QId):` we can stop a request created for the test without affecting the normal function of the component.
- Test cases with the `getSensorsInfo()` method do not affect the normal function of the component.
- With the `updateSensor(Sid Integer, String attr, Object nValue, int priority)` method, to ensure component correctness, before executing a test, we must safeguard the information of sensors, and after the testing process, the sensor information must be restored to the original state.
- With test cases that need the "idle" initial state, *i.e.* to affect the PQS component to *idle* state, we must stop all queries through this component. Requests that are created for testing do not interfere on normal functionality of the component. But in the case where there are queries of the system, we use the second option, *i.e.* this test case is abandoned and the component continues with other test cases.

D. Connection between the components

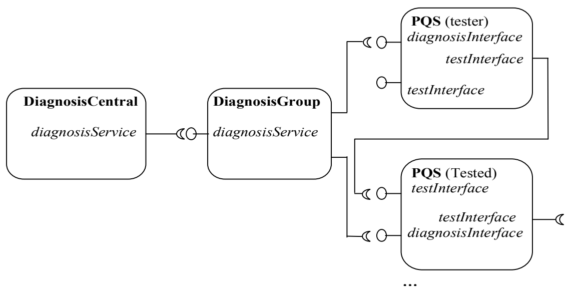


Figure 8. The connections between components

After the diagnosis request of the user from the graphic interface, the DiagnosisCentral component collects all information about the diagnosable components in the system. It divides the system into diagnosis groups and sends the user's diagnosis parameters to all groups. The DiagnosisGroup component schedules inter-component testing in the corresponding group according to the requested diagnosis algorithm. It provides tester component with the tested component identities and test cases. The tester component tests the tested component via the *testInterface* interface (Figure 8).

V. Experimental Results

For the experimentation, the considered heterogeneous sensor data management system comprises: 2 gateways, each of which has 7 PQSs. The PQS are supposed here to provide a temperature service. In each group, there is one faulty component. Each component is tested by 6 test cases.

In this case study, we adopt two diagnosis algorithms from the literature: a distributed diagnosis algorithm [10] and a centralized algorithm [4].

Table II below shows obtained measures of needed additional physical memory for diagnosis purposes. It can be concluded that, for these test cases, the diagnosis service does not have high demands on memory.

TABLE II. Additional Cost of Diagnosis Service

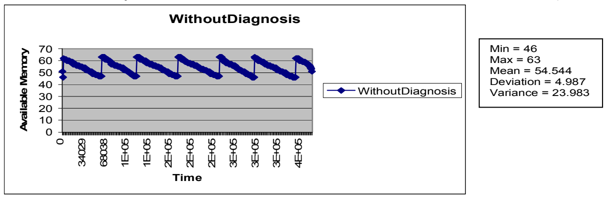
Criterion	**Additional cost of diagnosis service**	**Additional cost in each component.** **Depend on test cases (6 test cases)**	**Application**	**% compared with the application**
Compile classes size	27.4 kB	9.2 kB	503.49 kB	~ 7.27 %
Bundles size	23.82 kB	4.51 kB	5.063 MB	~ 1.1%

TABLE III. Centralized and Distributed Diagnosis

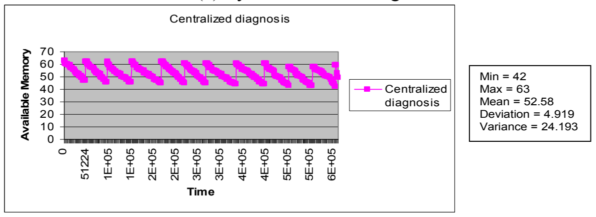
Criterion	**Centralized diagnosis method**	**Distributed diagnosis method**
Fault number (t)	3 ≥ t for each group 6 ≥ t for the whole system	6≥ t for each group 12≥ t for the whole system
Diagnosis Time (ms)	50331	171988
Test number	O(n)	O(n)

Table III compares obtained results for centralized diagnosis and distributed diagnosis. The sensor data management system can tolerate up to 6 faulty components in each group or 12 faulty components in the whole system with the distributed diagnosis algorithm. It can tolerate up to 3 faulty components in each group or 6 faulty components in the whole system with the centralized diagnosis algorithm. These results show that there should be a compromise (trade-off) in resource utilization between diagnosis methods. The distributed method can tolerate more faulty components, while the centralized diagnosis method needs less time to detect faults.

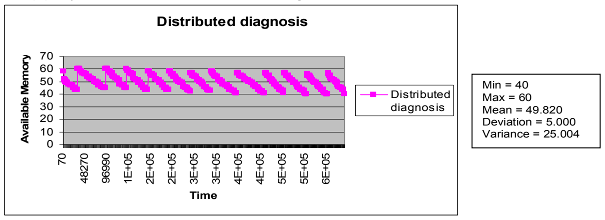
Figure 9 shows the measures of available memory of the application without diagnosis (9.a), with diagnosis but non-considering the available resources of the system (9.b and 9.c) and with diagnosis in considering the memory resource (9.d and 9.e). The tests are executed when the memory availability is above 40%. The memory is calculated as the difference between memory used by the application and the total memory dedicated to the Java Virtual Machine (~5M).



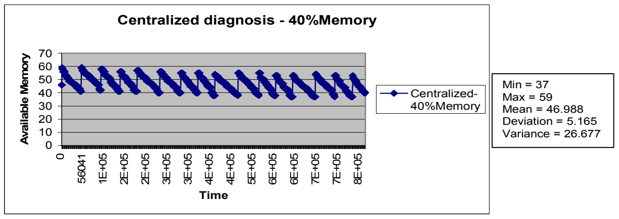
(a) System without diagnosis



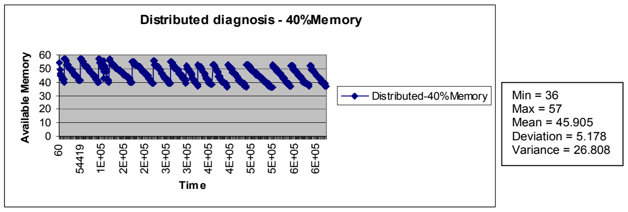
(b) System with centralized diagnosis and non-constrained resources



(c) System with distributed diagnosis and non- constrained resources



(d) System with centralized diagnosis and availability of 40% of memory resource



(e) System with distributed diagnosis and availability of 40% of memory resource

Figure 9. Available memory for the sensor data management system

The obtained results, illustrated in Figure 9, show that the additional cost for the diagnosis is very low. Indeed, the memory used by the diagnosis service is between 2 and 9% of the total memory.

VI. CONCLUSION

In this paper, we demonstrated the application of our diagnosis service on an industrial case study – a heterogeneous sensor data management system. We illustrated the implementation of our diagnosis service on this concrete case study in order to increase the reliability of the whole system. This diagnosis service provides the ability of detection and location of faulty components during runtime system execution.

The obtained results show that the additional cost for the diagnosis is low and allow us to consider that on-line diagnosis is a viable and interesting approach for handling faults in software component based systems. We are currently working on more experimental results while varying the number of faults, number of components, the test period, *etc*.

REFERENCES

[1] F. Favarim, J. Fraga and F. Siqueira, “Fault-tolerant CORBA Component”, IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 2003.

[2] V. Marangozova and D. Hagimont, “An Infrastructure for CORBA Component Replication”, 1st IFIP/ACM Working Conference on Component Deployment, Berlin, Allemagne, Juin 2002.

[3] iCMG, available at http://www.componentworld.nu/.

[4] F. P. Prerapata, G. Metz and R. T. Chien, “On the connection assignment problem of diagnosticable systems”, IEEE Transactions on Electronic Computers, vol. EC-16, n°6, pp. 848-854, December 1967.

[5] T.Q. Bui and O. Aktouf, “Diagnosis service for embedded software component based systems”, Proceedings of the Second International Workshop on Engineering Fault Tolerant Systems, pp. 14-19, Dubrovnik, Croatia, September 2007.

[6] T.Q. Bui and O. Aktouf, “On-line testing of software components for diagnosis of embedded systems”, Proceedings of the 4th International Conference on Embedded and Real-Time Computing Systems, pp.330-336, Volume 22, Prague, Czech Republic, July 2007.

[7] M. Barborak and M. Malek, “Partitioning for efficient consensus”, IEEE 26th Hawaii International Conference on System Technology Sciences, pp. 438-446, June 1993.

[8] D. Suliman, B. Paech, L. Borner, C. Atkinson, D. Brenner, M. Merdes and R. Malaka, "The MORABIT approach to runtime component testing", pp. 171-176, 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Chicago, September, 2006.

[9] A. Ketfi, N. Belkhatir, “A metamodel-based approach for the dynamic reconfiguration of component-based software”, The Eighth International Conference on Software Reuse, Madrid, Spain, July 2004.

[10] R. Bianchini and R. W. Buskens, “An adaptative distributed system level diagnosis algorithm and its implementation”, Proceedings of the 21st international IEEE Symposium on Fault-Tolerant Computing, IEEE CS Press, pp. 616-626, Montreal, Canada, June 1991.

[11] T.Q. Bui, O. Aktouf and M. Dang, “Software Component Diagnosis Service: Architecture Description”, Proceedings of the Third International Symposium on Industrial Embedded Systems, pp. 134-140 , La Grande Motte, France, June 2008.

[12] L. Gürgen, C. Labbé, C. Roncancio, A. Bottaro and V. Olive, “SStreaMWare : a service oriented middleware for heterogeneous sensor data management”, Proceedings of the 5th International Conference on Pervasive Services (ICPS), ACM Press, pp. 121-130, Sorrento, Italy, July 2008.

[13] OSGi, available at http://osgi.org.