

# Multi-fault Diagnosis in Dynamic Systems

*Natalia Odintsova, Irina Rish, Sheng Ma*  
*IBM T.J. Watson Research Center*  
*19 Skyline Drive, Hawthorne, NY 10532*  
*{nodints,rish,shengma}@us.ibm.com*

## Abstract

In this paper, we address the problem of diagnosing multiple faults in dynamically changing systems. Currently used popular techniques such as codebook and active probing suffer from limitations imposed by their "static", non-temporal nature, and single-fault assumptions. We propose a very simple, linear-time multifault scheme, capable of tracking system changes and diagnosing multiple faults much more accurately than previously used approaches. We provide empirical results demonstrating the advantages of our approach and analyze the effect of test set quality on the diagnostic accuracy.

## Keywords

real-time diagnosis, multi-fault diagnosis, change detection, event correlation

## 1. Introduction

In this paper, we consider the task of real-time event correlation and problem diagnosis. We present a novel approach that extends commonly used diagnostic techniques (such as codebook [5] and active probing [8]) to allow change detection and continuous monitoring of multiple failures in dynamically changing systems. Particularly, we focus on the following important issues.

*1. Change detection.* Currently existing approaches to event correlation and diagnosis typically assume that the state of the system does not change during diagnosis cycle, and thus the incoming events and measurements are treated as symptoms of same problem. Clearly, this assumption is often violated in dynamically changing environments where problems occur sequentially, and thus the symptoms arriving at different times may be inconsistent with each other. 'Static' approaches to diagnosis such as codebook method [5] are unable to detect such changes and treat inconsistent symptoms as 'noise' in observations, which results into diagnostic errors.

*2. Handling multiple faults.* In a large, multi-component system the probability of encountering multiple problems at a time (e.g., failures of several system components) increases with increasing number of components, assuming the probability of a single problem (e.g. fault of a component) is fixed. Thus, a typical "static" diagnostic engine (e.g., codebook or similar approaches) must be able to diagnose such multiple faults simultaneously. However, multiple-fault diagnosis in a system with large number of components is known to be computationally challenging; for example, using a codebook approach we will have to enumerate an exponential number of possible fault combinations and provide their symptoms (columns in the codebook table). Similar complexity issues are encountered by other approaches such as constrained-based multi-fault diagnosis (see [2])

and probabilistic methods using Bayesian networks: in general, multiple-fault diagnosis in such frameworks yields NP-hard inference problems. On the other hand, a diagnostic engine that tracks system's changes over time can handle multiple faults incrementally, one at a time, which yields significant computational savings (assuming, of course, that the changes happen at some reasonable frequency so that the diagnostic engine can process them sequentially; however, in general, multiple-fault situations cannot be avoided completely).

3. *Diagnostic capability of available measurements.* Finally, an important issue affecting the quality of diagnosis (i.e., its speed and accuracy) is the quality, or "diagnostic power" of the available symptoms, measurements, or events. Most of the current research on problem determination is focused on developing various diagnostic algorithms. However, we should not expect diagnosis to be accurate, no matter how sophisticated is the algorithm, unless we provide a set of measurements or events that have enough "diagnostic power", i.e. contain enough information about the problem (otherwise, we get a "garbage in, garbage out" situation). Also, the more problem combinations are possible in a system (e.g., due to multiple faults), the more powerful measurement set we may need. We will identify a phenomenon called *shielding* when one component's failure makes it impossible to obtain information about some other components ("shields" those components). For example, a failure of a router on the path to a server will shield the server, if all available measurements of web performance are end-to-end transactions (e.g., ping, web-page access) going through this router. This paper focuses on the issues described above and makes several contributions:

1. first, standard codebook approach is extended to a simple, linear-time multifault algorithm (*generic multifault*) that returns an upper bound (a superset) of faulty nodes; the approach is still 'static' as it does not handle system changes;
2. next, generic multifault is further extended to track faults and repairs in a dynamic system; the resulting *sequential multifault* algorithm can efficiently handle multifault situations, significantly reducing diagnostic error of codebook approach, while at the same time avoiding exponential complexity of common multifault diagnosis approaches;
3. finally, we provide an empirical study demonstrating the advantages of the proposed algorithms over commonly used "static" approaches (such as codebook and active probing), and analyze empirically how the measurement set selection affects the quality of diagnosis. Our evaluation on both synthetic and real-life problems demonstrate that, in dynamic systems, the proposed method is significantly more accurate than its competitors.

## 2. Current Approaches and Their Limitations

We adopt a commonly used problem-determination framework known as *codebook* [5] (also called *dependency matrix* [1, 8]). The codebook, or dependency matrix,  $D = [d_{ij}]$  is a 0/1 matrix where columns correspond to possible problems, and rows correspond to observations (events, measurements). In the matrix,  $d_{ij} = 1$  if problem  $p_j$  causes event  $e_i$  to happen, and  $d_{ij} = 0$  otherwise. Assuming the list of problems is complete, i.e. there are no other unknown problems that could cause same events to happen in the absence

of given problems (so-called "closed world assumption"), we can use the combinations of events corresponding to column  $j$  as a *symptom*, or *codeword*, for the problem  $p_j$ . The problems are uniquely diagnosable if all columns are different.

The dependency matrix can be also interpreted as a collection of propositional formulas, one per row, of the form  $e_i \rightarrow p_{j_1} \vee \dots \vee p_{j_k}$ , where  $j_1, \dots, j_k$  correspond to nonzero entries in row  $i$ , i.e. to the problems affecting event  $e_i$ . This interpretation is natural in the context of diagnosis by *probing* [1, 9, 8] where columns correspond to particular components in a system (e.g. nodes in a computer network, such as routers and servers, or software components such as web- and database applications), and rows correspond to end-to-end transactions, called *probes*, which involve ("go through") subsets of system components. Boolean variable  $p_j$  denotes a problem, or failure, at component  $j$  ( $p_j = 1$  if component failed and  $p_j = 0$  otherwise), and event  $e_i$  denotes outcome of a probe  $i$  ( $e_i = 1$  if probe fails and 0 otherwise). Clearly, a probe is a disjunctive test, since it fails if and only if at least one of its components fails. For example, a transaction "open web page" is successful if and only if the web server is OK and all routers are OK on the path from *probing station*, where the transaction is initiated, to the web server (again, assuming there are no other unexpected problems - otherwise they must be included in dependency matrix or treated as "noise"). Let us consider a simple example in Figure 1a. A network on the left contain five nodes, where two nodes (X1 and X2) are probe stations, nodes X4 and X5 are servers, and node X3 is a router; the dependency matrix on the right shows a set of available probes, e.g. web page request going from X1 to X5, SQL query going from X2 to X4, and direct tests of probe stations themselves. Note that all columns are distinct, and thus any single fault is uniquely diagnosable.

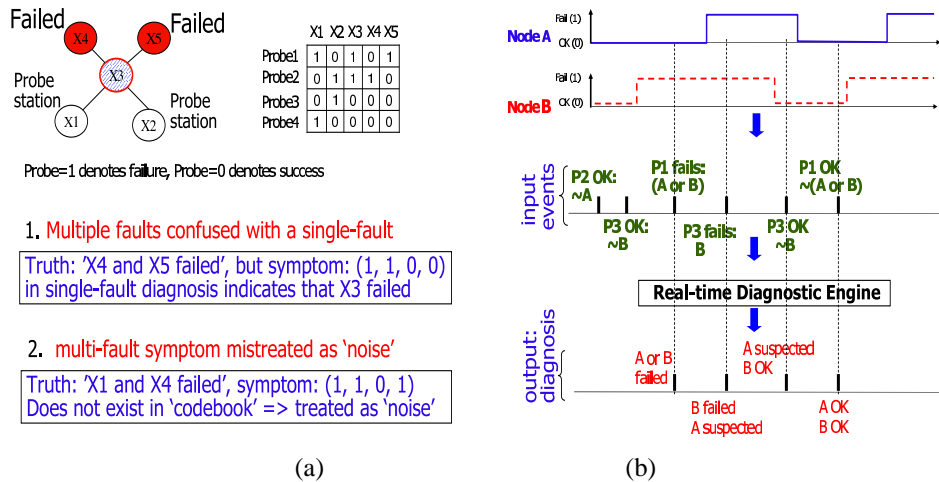
There are two kinds of diagnostic approaches that are based on dependency matrix, depending on whether the observations (events, probes) are passively observed, or actively selected. The original *codebook* algorithm [5], as well as earlier work on probing [1, 9], passively observe all available events or probe outcomes, and match the vector of observations to the columns of the matrix (if there is no match, the codebook approach returns the closest column in Hamming distance). Recently, a more efficient version, called *active probing* [8], has been proposed. Instead of gathering all available probes, this method selects probes on demand, choosing most-informative next probe depending on the outcomes of the previously observed probes. Active probing often demonstrates significant savings, often up to 70%, in the total number of probes and diagnosis time. However, both active and passive approaches assume that: (a) there is a single problem (single failed component); (b) no changes occur in a system while the events or probes are collected.

## 2.1 Limitations of Current Approaches

As any approach, codebook and probing techniques have their limitations due to assumptions the make, such as single-fault and static system, which lead to diagnostic errors.

### *Handling Multiple Faults*

Let us consider several examples showing limitations of codebook and similar approaches in multifault scenarios. First, a combination of multiple faults may produce a symptom that will be confused with another single fault symptom. Consider the network



**Figure 1:** (a) A simple network and its dependency matrix; several examples of multi-fault situations that cause errors in codebook approach. (b) Dynamically changing system: sequential faults and repairs cause changes in probe outcomes and must be detected and properly diagnosed.

in Figure 1a and assume that two nodes, X4 and X5 fail. Then both Probe1 and Probe2 fail, yielding a symptom vector (1,1,0,0). However, this is the symptom of X3 failure, i.e., the intersection of two failed probes, and there is no way to distinguish its failure from simultaneous failure of X1 and X2 given the current probe set. Thus, codebook approach will make three mistakes: miss two faults (2 false-negative errors) and blame "innocent" node X3 (1 false-positive).

Another erroneous situation occurs when a multifault symptom simply does not exist in the codebook, as for example, column (1,1,0,1) that corresponds to failure of X1 and X4. Codebook approach will again return X3 as a failure, since its symptom is the closest to (1,1,0,1) in Hamming distance.

It can be argued that we simply should include columns corresponding to all possible fault combinations. But this is clearly intractable since there are up to  $2^n$  different fault combinations in a system with  $n$  components. Another problem is that not all combinations of probe outcomes are even realizable, for example due to topological and other constraints. Thus, some multifault situations can be inherently unrecognizable because some components may become "shielded" by the failures of other components.

**Definition:** A component  $X$  is shielded by the failure of the component  $Y$  if all probes going through  $X$  go through  $Y$  as well.

For example, in Figure 1a, failure of node X3 shields both X4 and X5, and there is no way to find out if both X4 and X3 failed, or just X3. However, assume that X4 failed before X3; an algorithm capable of tracking system changes would diagnose X4 failure

first, and *then* diagnose failure of X3 when it happened (of course, if X3 failed before X4, even such sequential diagnosis would not be able to guarantee correct diagnosis of X4 since it will be already shielded by X3).

Various approaches to multifault diagnosis has been considered, especially in AI literature. A "classical" model-based approach views diagnosis as logical inference, or, in general, a constraint satisfaction problem, where diagnosis consists in finding a satisfying assignment to system's components (indicating OK or failure state), given a set of constraints describing the system (e.g., in our case the set of propositional formulas implied by the dependency matrix), and a set of observations (e.g. see [2]). Similarly, a probabilistic approach, such as Bayesian networks [6], describes a probabilistic model of a system and uses it to find most-likely diagnosis. However, the problem with these type of approaches is their computational complexity, since, in general, both logical and probabilistic diagnosis are NP-hard problems. Thus, handling multiple faults usually requires some kind of approximation [7, 10].

#### *Handling system changes*

Another limitation of existing approaches is their inability to track changes which results into higher errors due to possibly contradictory observations collected before and after the change. For example, consider a very simple dynamic system containing two components, A and B (see Figure 1b), and 3 probes: Probe1 is a transaction involving both components (e.g., opening a web page on a web server which requests data from a database server), while Probe2 and Probe3 are *ping* commands testing server A and server B, respectively. Initially, both A and B are OK ( $A=0$  and  $B=0$ ), and the Probe2 and Probe3 executed at this time both return OK. Next, node B fails; Probe1 that was executed after the B's fault returns failure, which corresponds to  $(A \vee B)$  and contradicts previous observations. Diagnostic engine must detect this contradiction, identify the change in the system and report it (i.e., 'A or B failed'). Clearly, in realistic scenarios, there could be multiple probes involving different intersecting subsets of nodes, so that change detection will require more complex inference.

Note that existing approaches to multiple fault diagnosis in communication networks and distributed computer systems, such as those using Bayesian networks [3, 7, 10] and other probabilistic dependency models [4], typically do not track system changes and therefore may have an unnecessarily high error due to contradictory evidence.

### **3. Our Approach**

In this section we describe two multifault algorithms: a very simple *generic multifault* and its extension to more advanced *sequential multifault* algorithm for dynamic systems.

#### **3.1 Generic Multifault**

This algorithm makes no assumptions on the number of faults in the system. However, it still assumes that the system is static - that is, no changes happen during the diagnosis cycle, and thus all probe results must be consistent with each other. Given the dependency matrix and the probes outcomes, the algorithm proceeds as follows:

1. Find OK nodes: these are all the nodes through which at least one OK probe passed.

2. Find failed nodes: these are the nodes through which any failed probe passed such that all other nodes on its path are OK nodes, as determined in step 1.
3. Find *shielded* nodes: these are the nodes through which no probe goes other than those that go through any of already failed nodes. Thus, all those probes will return 'failure' and it is impossible to determine the state of such nodes, or, in other words, the nodes are 'shielded' by the failures of nodes found in step 2.
4. The remaining nodes are "possible failures", in the sense that certain combinations of their failures can produce the given set of probe outcomes. In principle, we can enumerate all such combinations. However, it may be computationally complex, and impractical. In our implementation, we simply report all these nodes as (possible) failures.

We illustrate this algorithm on a simple example. Suppose we have the following dependency matrix and set of probe outcomes (here we use "1" to indicate probe's failure, and "0" for probe's success):

	A	B	C	D	E	F	Probe outcomes
Probe 1	1	1	0	0	0	0	1
Probe 2	0	1	1	1	0	1	1
Probe 3	1	0	1	0	0	0	0
Probe 4	0	0	0	0	1	1	1

1. Since probe 3 is OK, nodes A and C are OK. There are no more OK probes here.
2. We now remove rows and columns corresponding to probe 3, and nodes A and C:

	B	D	E	F	Probe outcomes
Probe 1	1	0	0	0	1
Probe 2	1	1	0	1	1
Probe 4	0	0	1	1	1

Since probe 1 failed, and has only one node on its path, this node (B) has failed.

3. Further, removing column B and the rows corresponding to the probes going through B, we get:

	D	E	F	Probe outcomes
Probe 4	0	1	1	1

Now node D doesn't have any probes going through it that would not go through B, so D becomes shielded by B's failure.

4. The remaining nodes E and F are possible failures. The outcome of probe 4 can be explained by failure in E, or in F, or in E and F together. We will add E and F to the set of failed nodes. Thus, the output in this example will be: OK nodes: A, C; Failed nodes: B, {E, F}; Shielded nodes: D.

Without any assumptions on the number of failures in the system, we cannot make any further conclusions given the current probe observations. Clearly, the accuracy of the algorithm is affected by the quality of the available probe set. If the shielded nodes are interpreted as failures, the algorithm's false-negative error (missed faults) is zero. Its false-positive error (OK nodes misdiagnosed as failures), however, can be high if a lot of shielding occurs in the system. Constructing the probe set so that to minimize shielding of nodes by other nodes' failure will significantly improve the algorithm's performance.

Generic multifault that reports shielded nodes as failed does not miss any faults, although its false-positive error (the amount of OK nodes reported as faulty) can be high

for certain dependency matrices. We will refer to this algorithm as "safe", oppose to "non-safe" that reports shielded nodes as OK. In fact, the generic multifault algorithm acknowledges the shielded nodes, and it is up to the user to decide how to interpret them. We will show in empirical section that, depending on the probability of fault in a system, we should prefer "safe" or "non-safe" version.

Due to its simplicity, the generic multifault algorithm has low computational complexity – it is linear in the number of probes, and the number of nodes, while other commonly used algorithms such as Bayesian inference and constraint satisfaction approaches can be worst-case exponential in the number of nodes. However, there is no "free lunch": while such exact approaches can find the most probable combination of faults, or all minimal fault combinations explaining the probe outcomes, the simplistic generic multifault only finds a superset containing all faulty nodes (an 'upper bound') and may have a high false positive error. Also, the algorithm assumes that the system did not change while the probe outcomes were obtained, and thus there are no possible contradictions in the probe outcomes. In other words, the algorithm still cannot handle system changes.

### 3.2 Sequential Multifault

In this section, we extend the generic multifault approach to handle dynamically changing system. This algorithm is still linear in the number of probes, and yet it is able to diagnose multiple-fault situations. Unlike generic multifault algorithm, the sequential multifault algorithm handles dynamic environment by keeping track of changes in the system, as they occur. The algorithm does not assume the incoming probe results are consistent with each other; moreover, such inconsistency helps to detect a change in the system. The algorithm does not restrict the amount of faulty components in the system, but it assumes that only one change can happen at a time (i.e., failure or repair of one component), and that processing of each change is fast enough so that no other change occurs while the current change is being processed. Such assumptions allow computationally efficient processing of multiple failures in the system by applying single-fault diagnosis approach to localize each of the faults sequentially. At a very high-level, the algorithm performs the following monitoring loop:

```
initialize-system-state;
while (true)
    if current observation contradicts previous observations {
        diagnose change;
        report results;
    }
```

Particularly, the algorithm monitors changes in the system's states using two sets of probes: set for fix (i.e., repair) detection to monitor nodes that are known as failed, and set for failure detection to monitor nodes that are known to be OK. If no change in the system has occurred, the probes from the first set are expected to continue returning "failure", whereas the probes from the second set are expected to continue returning OK. A probe outcome different from the one expected indicates a change in the system. When the algorithm detects a change, it diagnoses (locates) the changed component. Since the algorithm tracks the changes sequentially, it requires to be given

an initial system state. If the initial system state is not known, it can be determined by applying the generic multifault algorithm. After a change has been located and processed, the algorithm updates its set of measurements - probe sets for fix and failure detection. It also determines the set of shielded nodes - the nodes that are shielded by the current set of failures. Below is the pseudocode for the sequential multifault algorithm.

### Sequential Multi-Fault (SMF) algorithm

---

Input: Dependency matrix, probe outcomes as they arrive,  
initial system's state  
Output: diagnosed system state

```

Initialize nodes according to the initial system state
while (true) { // endless loop, i.e. continuous monitoring
    shielded nodes = shielded(set of current failures);
    probesForFixDetection F = setForFixDetection();
    probesForFailureDetection D = setForFailureDetection();

    change = "no change"
    while(change = "no change") {
        receive outcome of a probe P
        if(outcome(P) = OK and P belongs to F) change = "repair"
        else if (outcome(P) = FAILED and the P belongs to D)
            change = "failure"
    } //end while
    if (change = "repair") {
        move all nodes belonging to P to OK set
    } else if (change="failure") {
        fault = DiagnoseSingleFault(probe)
        move fault to the fail set
    } // end if
    Output: OK nodes, failed nodes, shielded nodes
}

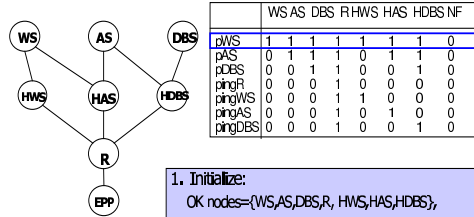
```

---

The shielded nodes are determined by removing from the dependency matrix the rows corresponding to the probes that go through any of the failed nodes. These are the probes that constitute probe set for failure detection - the expected outcome of these probes is 'failure', and their success indicates a change in the system (i.e., repair of a previously failed node). This set can be further optimized - for example, if a failed node has later become shielded by another node's failure, we will not be able to detect its repair until the node that caused the shielding is repaired. Thus, we may monitor only this cause, and then check the status of the shielded node after it becomes reachable again.

Next, given the remaining probes (i.e., the ones left after excluding probes going through any of the failed nodes), we use a greedy search approach in order to select a

**Initial state: no failures**



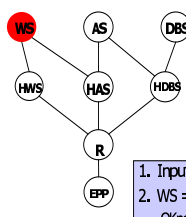
	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Initialize:  
OK nodes={WS,AS,DBS,R, HWS,HAS,HDBS},  
FaultyNodes = {}, Shielded = {}
2. Probe set for FIX detection = {}  
Probe set for FAILURE detection D={pWS}
3. While (no failure in D) {}

**Output: no alarm message sent**

(a)

**Failure 1: WS failed**



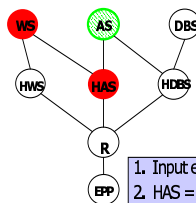
	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Input event: pWS failed
2. WS = DiagnoseSingleFault(pWS)  
OKnodes={AS,DBS,R,HWS,HAS,HDBS}  
FaultyNodes = {WS}, Shielded = {}
3. Probe set for FIX detection: F = {pWS}  
Probe set for FAILURE detection: D={pAS, pingWS}
4. While (no failure in D & no repair in F) {}

**Output: WS failed**

(b)

**Failure 2: HAS failed**



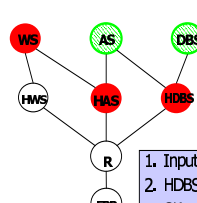
	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Input event: pAS failed
2. HAS = DiagnoseSingleFault(pAS)  
OKnodes={DBS,R,HWS,HDBS},  
Shielded={AS}, FaultyNodes = {WS,HAS}
3. Probe set for FIX detection F = {pingAS}  
Probe set for FAILURE detection: D={pDBS, pingWS}
4. While (no failure in D & no repair in F) {}

**Output: HAS failed**

(c)

**Failure 3: HDBS failed**



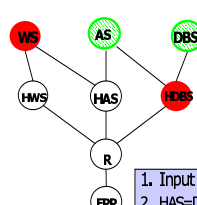
	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Input event: pDBS failed
2. HDBS = DiagnoseSingleFault(pDBS)  
OKnodes={R,HWS}, FaultyNodes = {WS,HAS,HDBS},  
Shielded={AS,DBS}
3. Probe set for FIX detection: F = {pingAS, pingDBS}  
Probe set for FAILURE detection: D={pingWS}
4. While (no failure in D & no repair in F) {}

**Output: HDBS failed**

(d)

**Repair 1: HAS repaired**



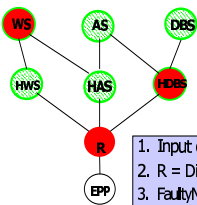
	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Input event: pingAS returns OK
2. HAS=DiagnoseRepair(pingAS)  
FaultyNodes = {WS,HDBS}, OKnodes={R,HWS,HAS},  
Shielded={DBS,AS}
3. Probe set for FIX detection: F = {pingDBS}  
Probe set for FAILURE detection: D={pingWS, pingAS}
4. While (no failure in D & no repair in F) {}

**Output: HAS repaired**

(e)

**Failure 4: R failed**



	WS	AS	DBS	R	HWS	HAS	HDBS	NF
pWS	1	1	1	1	1	1	1	0
pAS	0	1	1	1	0	1	1	0
pDBS	0	0	1	1	0	0	1	0
pingR	0	0	0	1	0	0	0	0
pingWS	0	0	0	1	1	0	0	0
pingAS	0	0	0	1	0	1	0	0
pingDBS	0	0	0	1	0	0	1	0

1. Input event: pingWS failed
2. R = DiagnoseSingleFault(pingWS)
3. FaultyNodes = {R,HWS,HDBS}, OKnodes={},  
Shielded={HWS,AS,HAS,DBS}
4. Probe set for FIX detection: F = {pingR}  
Probe set for FAILURE detection: D={}-no probes left!
5. While (no repair in F) {}

**Output: R failed**

(f)

**Figure 2:** Illustration of sequential multifault algorithm.

minimal subset of probes for failure detection. If there is no change in the system, we expect these probes to be OK. If at least one of them fails, indicating that a new fault occurred in the system, we can diagnose (localize) the fault by using any single-fault diagnosis algorithm, such as, for example, codebook or active probing (which will be called by subroutine `DiagnoseSingleFault()` in the pseudocode) on the remaining subset of the dependency matrix - that is, on the matrix that results after crossing out columns corresponding to failed nodes and rows corresponding to all the probes going through any of the failed nodes. In our implementation, we use active probing algorithm; as shown before [8], active probing can significantly reduce the average number of probes required to localize a fault when compared with passive probing, or codebook approach.

We now give an example illustrating the sequential multifault algorithm (see Figure 2). In this example, we assume that there is no failed nodes in the initial system state.

1. *All nodes are functioning properly; all probes return OK* (Figure 2a).

Since there are no failures at the current state, probe set for fix detection is empty. Probe set for failure detection consists of the longest probe that covers all nodes, pWS.

2. *Node WS failed, probe pWS failed* (Figure 2b).

We locate the fault (WS) by active probing algorithm, update the diagnosed system's state, and delete WS and all probes going through it from the dependency matrix. We see that the failure of WS doesn't shield any nodes, because all remaining nodes still have some 1's in their columns.

3. *Node HAS failed, probe pAS returns failed* (Figure 2c).

After locating the fault (HAS) and modifying the dependency matrix, we see that HAS's failure has shielded node AS. Note also, that the failed node WS has also become shielded by this failure. So we don't include the probe pWS in the set for fix detection. Rather, we can check the status of WS after HAS has got repaired.

4. *Node HDBS failed, probe pDBS failed* (Figure 2d).

The faulty node in this case is HDBS. Its failure shielded one more node - DBS. Probe set for failure detection consists of a single probe - pingWS - that covers both of the remaining OK nodes (HWS and R). For fix, we monitor only HAS and HDBS; the failed node WS is shielded.

5. *Node HAS repaired, probe pingAS returns OK* (Figure 2e).

Success of probe pingAS indicates that HAS has repaired. However, we see that none of the shielded nodes became unshielded, because they all are still shielded by HDBS. Node WS is also shielded, so we don't include pWS into the probe set for fix detection.

6. *Node R failed, probe pingWS returns failure* (Figure 2f).

Active probing finds that R is down. R's failure has shielded all the nodes. Now the probe set for failure detection is empty - we don't have any probe not going through a failed node. Probe set for fix detection consists of the single probe - pingR, that monitors node R for repair. We say that R is the bottleneck, because we cannot update information about other nodes' state before R has got fixed.

The sequential multifault algorithm has both false-positive and false-negative errors. It assumes the shielded nodes are OK unless they had failed before they became shielded. Clearly, this may lead to a false-negative error (i.e. some faults will be missed if they occurred at shielded nodes). In our example, after R is repaired, the shielded nodes HWS

and HAS will return to the OK set, when in fact they may have failed while R was down. Also, if the probe set is insufficient, certain combinations of faults can result in the remaining dependency matrix that is used to locate the next (single) fault (i.e., the matrix that remains after removing faulty nodes and all the probes going through them) that does not provide unique diagnosis (that is, the matrix that has two or more identical columns). In case when one of the undistinguishable nodes is actually down, declaring the whole group of undistinguishable nodes as failures yields false-positive error.

An alternative approach is to treat all shielded nodes as faulty; this will eliminate the false-negative error, i.e. no fault will be ever missed. We will call this version "safe" sequential multifault, as opposed to "unsafe" version described above. The price we pay for zero false-negative error is an increase in the false-positive error, which may become significant when faults are rare, and when there are many shielded nodes.

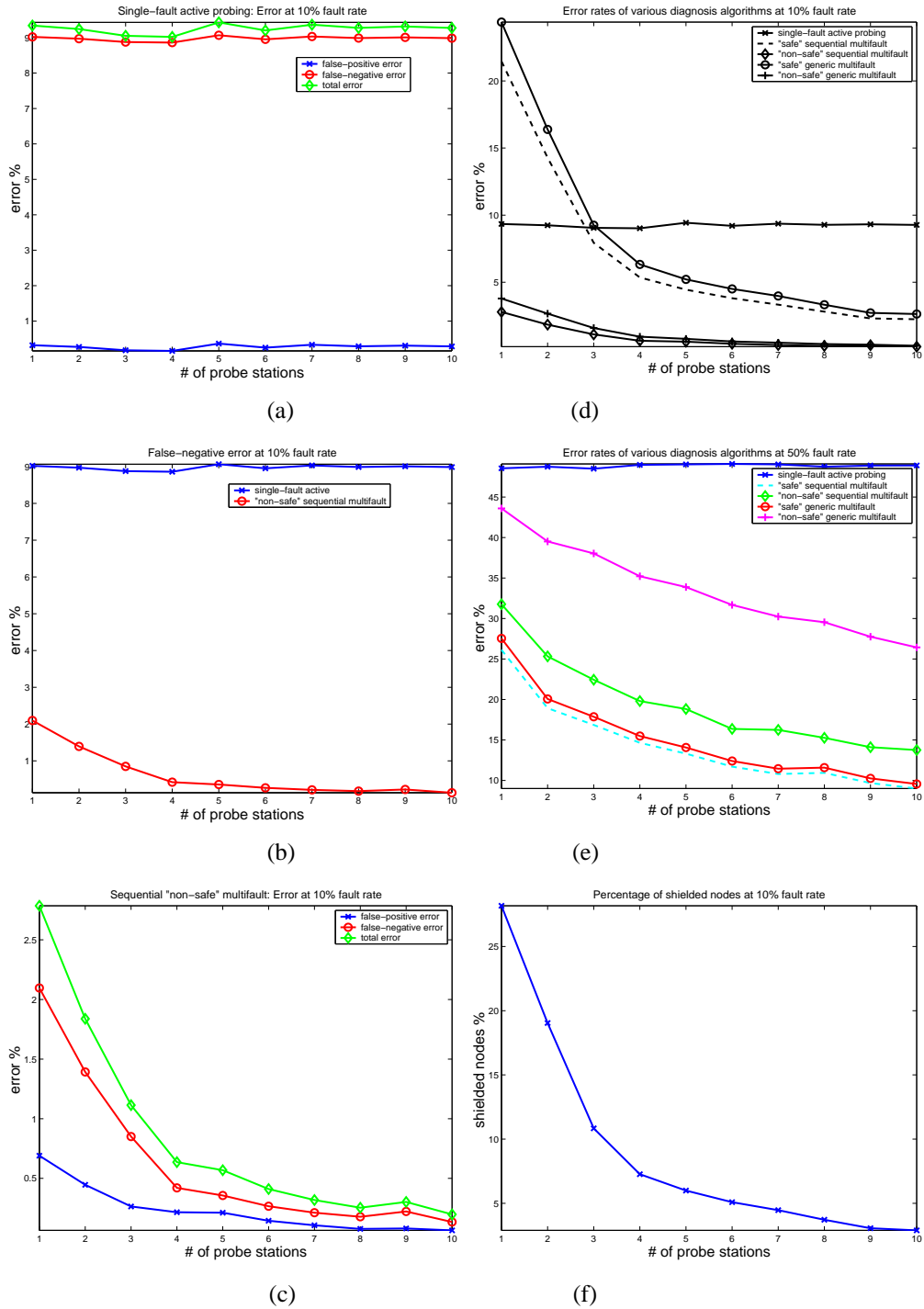
#### 4. Empirical Results

In this section, we evaluate the performance of both generic and sequential multifault algorithms, and compare these algorithms with the single-fault active probing approach (which is practically equivalent to codebook in diagnostic ability, but is more efficient due to active probe selection). The experiments are performed on randomly generated networks with 50 nodes and on a real-life router-level network obtained from a large company. We vary the number of probe stations, also called *sources*, and expect that increasing number of sources will decrease the amount of shielded nodes, since nodes may become reachable from multiple sources, and thus improve the accuracy of diagnosis.

For synthetic networks, the probe sets were constructed by randomly choosing sources and then constructing shortest paths from each source to every other node. The number of probe stations varied from 1 to 10. The results were averaged over 30 trials for each number of sources. The simulations were run on sequences of 100 consecutive changes in the nodes' states, which were generated randomly, one change at a time (i.e., at each step, only one node can change its state). At each step, the change type - failure or repair - was chosen according to a "fault density" parameter that took values from 0 to 1 (fixed for each sequence, independent on the current system's state), which allowed to vary the average number of faults in the system over the whole sequence. We present results for the fault densities 10% and 50% (that is, when 10% and 50%, correspondingly, of all nodes were down on average).

The error of the single-fault algorithm is close to the number of faults in the system. If there are  $N$  faults in the system, the single-fault algorithm will either find one of them, and miss all the others, or miss all of them, and instead diagnose some OK node as failure (as in example shown in Figure 1a). In the first case, its false-negative error is  $N-1$ , with positive error being zero. In the second case, its false-negative error is  $N$ , and false-positive error is 1. So the lower bound on average total error is  $N-1$ , and the upper bound is  $N+1$ , where  $N$  is the number of actual faults in the system. This estimation holds for any implementation of the single-fault algorithm (active probing or codebook).

Suppose nodes A and C are down. Both probes return failure. Since the closest match is column B, node B will be misdiagnosed as a failure. As we can see in Figure 3a, the false-



**Figure 3:** Left column: error rates of the single-fault algorithm versus "non-safe" sequential multifault algorithm for the 10% fault density: (a) single-fault algorithm; (b) false-negative error of the both algorithms; (c) sequential multifault algorithm. Right column: error rates of the different algorithms (d) at 10% fault density and (e) 50% fault density; (f) the fraction of shielded nodes at 10% fault density.

negative component constitutes the larger part of the total error for the single-fault active probing algorithm. Increase in the number of sources does not yield any error reduction.

Sequential multifault (in its "non-safe" version) also has both error components, although it misses notably fewer faults than the single-fault algorithm does (Figure 3b). Also, unlike single-fault, increasing the number of sources allows to reduce both false-negative and false-positive errors (Figure 3c).

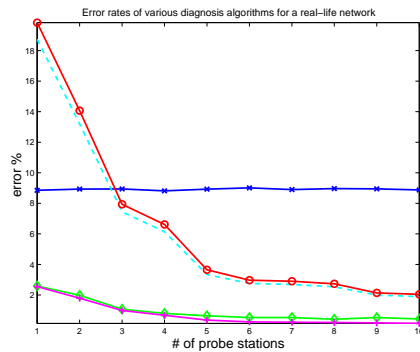
Choosing "safe" or "non-safe" version of multifault algorithms should depend on the anticipated fault density. When the fault density is relatively low (10% in our case), the "non-safe" sequential algorithm yields the best results (Figure 3d) with the total error rate more than 3 times less than that of the single-fault algorithm, even for the case of a single source. The error decreases even more with the increase in the number of sources. The error rate of the "safe" multifault algorithms is relatively high when the number of sources is small, but, unlike single-fault and "non-safe" multifault algorithms, "safe" multifault algorithm does not miss any faults (its false-positive error is zero). Increasing the number of sources to four produced the drastic error reduction, making the error of the "safe" multifault algorithms lower than that of the single-fault algorithm, whose error, moreover, is mostly composed of the missed faults (that is, whose false-negative error is high). In fact, the error curves for "safe" multifault algorithms mimic the change in the amount of shielded nodes with the increase in number of sources (Figure 3e). This suggests that constructing the probe set so that to minimize the shielding of nodes by other nodes' failure will help significantly improve the resolution of the "safe" multifault algorithms.

When the fault density is high (50% in our case), any of the multifault algorithms performs better than the single-fault algorithm (Figure 3f), with the "safe" versions yielding the best results. So at such fault rates it is better to interpret the shielded nodes as failed, rather than OK. It is interesting to note that, unlike low fault density, at high fault densities sequential "non-safe" multifault produces notably lower error than generic "non-safe" multifault algorithm. This difference can be explained by the fact that the sequential algorithm keeps track of the fault history, so if a node had failed and then later became shielded, sequential algorithm would "remember" this failure, rather than "blindly" declare all shielded nodes as OK, as the "non-safe" version of the generic multifault algorithm would do.

Finally, Figure 4 demonstrates the results for a real-life network at a large company. The network contains 65 nodes at a router level. We simulated a sequence of faults at 10% fault rate (i.e. 6.5 faults on average), and compared both "safe" and "non-safe" versions of generic and sequential algorithms versus single-fault active probing. The results are very similar to the ones obtained for random networks at 10% fault rate, which is not too surprising as it turns out the structure of this network (e.g., node degree distribution) was close to the structure of a random graph, and we used same fault rate of 10% for both simulations.

## 5. Conclusions

In this paper, we address the problem of diagnosing multiple faults in dynamically changing systems. Currently used popular techniques such as codebook and active probing suffer from limitations imposed by their "static", non-temporal nature, and single-fault



**Figure 4:** Error rates of different algorithms on a real-life network.

assumptions. We propose a very simple, linear-time multifault scheme, capable of tracking system changes and diagnosing multiple faults much more accurately than previously used approaches. We provide empirical results demonstrating the advantages of our approach and analyze the effect of test set quality on the diagnostic accuracy.

## References

- [1] M. Brodie, I. Rish, and S. Ma. Optimizing probe selection for fault localization. In *Distributed Systems Operation and Management*, 2001.
- [2] J. de Kleer and B.C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1), 1987.
- [3] JF. Huard and A.A. Lazar. Fault isolation based on decision-theoretic troubleshooting. Technical Report 442-96-08, Center for Telecommunications Research, Columbia University, New York, NY, 1996.
- [4] I.Katzela and M.Schwartz. Fault identification schemes in communication networks. In *IEEE/ACM Transactions on Networking*, 1995.
- [5] S. Kliger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Intelligent Network Management (IM)*, 1997.
- [6] J. Pearl. *Probabilistic Reasoning in Intelligent Systems*. Morgan Kaufmann, 1988.
- [7] I. Rish, M. Brodie, and S. Ma. Accuracy vs. efficiency trade-offs in probabilistic diagnosis. In *Proceedings of the The Eighteenth National Conference on Artificial Intelligence (AAAI2002)*, Edmonton, Alberta, Canada, 2002.
- [8] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik. Real-time Problem Determination in Distributed Systems using Active Probing. In *Proceedings of 2004 IEEE/IFIP Network Operations and Management Symposium (NOMS 2004)*, Seoul, Korea, 2004.
- [9] Irina Rish, Mark Brodie, and Sheng Ma. Intelligent probing: a Cost-Efficient Approach to Fault Diagnosis in Computer Networks. *IBM Systems Journal*, 41(3):372–385, 2002.
- [10] M. Steinder and A. S. Sethi. End-to-End Service Failure Diagnosis Using Belief Networks. In *Proceedings of Network Operations and Management Symposium (NOMS)*, Florence, Italy, 2002.