Gestalt: Fast, Unified Fault Localization for Networked Systems

Radhika Niranjan Mysore, *Google;* **Ratul Mahajan,** *Microsoft Research;*
Amin Vahdat, *Google;* **George Varghese,** *Microsoft Research*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/mysore

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Gestalt: Fast, unified fault localization for networked systems

Radhika Niranjan Mysore[1], Ratul Mahajan[2], Amin Vahdat[1] and George Varghese[2]

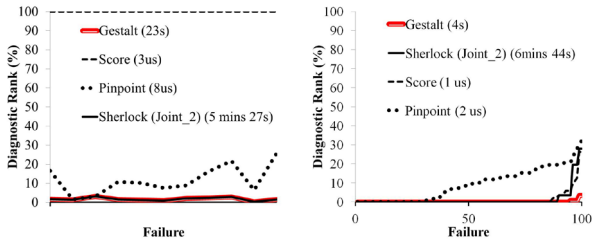
[1]Google [2]Microsoft Research

Abstract— We show that the performance of existing fault localization algorithms differs markedly for different networks; and no algorithm simultaneously provides high localization accuracy and low computational overhead. We develop a framework to explain these behaviors by anatomizing the algorithms with respect to six important characteristics of real networks, such as uncertain dependencies, noise, and covering relationships. We use this analysis to develop Gestalt, a new algorithm that combines the best elements of existing ones and includes a new technique to explore the space of fault hypotheses. We run experiments on three real, diverse networks. For each, Gestalt has either significantly higher localization accuracy or an order of magnitude lower running time. For example, when applied to the Lync messaging system that is used widely within corporations, Gestalt localizes faults with the same accuracy as Sherlock, while reducing fault localization time from days to 23 seconds.

1. Introduction

Consider a large system of components such as routers and servers interconnected by network paths. This system could be for audio, video, and text messaging (e.g., Lync [2]), for email (e.g., Microsoft Exchange), or even for simple packet delivery (e.g., Abilene). When transactions such as connection requests fail, a *fault-localization* tool helps identify likely faulty components. An effective tool allows operators to quickly replace faulty components or implement work-arounds, thus increasing the availability of mission-critical networked system.

As an example, we conducted a survey of call failures in the Lync messaging system deployed inside a large corporation. We found that the median time for diagnosis, which was largely manual, was around 8 hours because the operators had to carefully identify the faulty component from a large number of possibilities. This time-consuming process is frustrating and leads to significant productivity loss for other employees. A good fault localization tool that can identify a short list of potential suspects in a short amount of time would greatly reduce diagnosis time. Later in the paper, we will show how our tool, Gestalt, reduces by 60x the number of components that an operator must consider for diagnosis; and it has a median running time of under 30 seconds.



(a) Lync (real failures) (b) Exchange (simulated failures)

Figure 1: Applying different algorithms to two systems. Legend shows median time to completion.

Of course, we are not the first to realize the importance of fault localization, and other researchers have developed many algorithms (e.g., [3, 6, 8, 11, 13, 14, 16–18]).

However, we have consistently heard from operators (e.g., at Google and Microsoft) that the effectiveness of existing fault localization algorithms in terms of running times and accuracy depends on the network. There are no studies that connect network characteristics to the choice of algorithm, making it difficult to determine an appropriate algorithm for a given network. Figure 1 illustrates this difficulty by running three prior algorithms on two different networks. We picked these algorithms because they use disparate techniques. In the graphs, the *y*-axis is the diagnostic rank, which is the percentage of network components deemed more likely culprits than the components that actually failed; thus, lower values are better[1]. The failures are sorted by diagnostic rank. We

[1]In information retrieval terms, diagnostic rank includes the

provide more experimental details in §9,

The left graph shows the results for the Lync deployment mentioned above. We see that the algorithms perform differently. Sherlock [6](modified) does best, and SCORE [17] does worst. The right graph shows the results for simulated failures in an Exchange deployment [9]. We see that the algorithms exhibit different relative performance. SCORE matches Sherlock, and Pinpoint does worst. Further, the appropriate approach for the two networks differs—Sherlock for Lync, and SCORE for Exchange as it combines high localization accuracy and fast running time.

There is also a tradeoff between localization accuracy in the presence of impairments such as noise and computational cost for large networks. This tradeoff can be seen in Figure 1. While SCORE runs in a few microseconds, it localizes faults poorly for Lync. On the other hand, while Sherlock [6] has good performance for both networks, it takes a long time. In large networks, this time can be days. Running time matters because it directly influences time to recovery. For a large network like Lync, ideally we would want a localization algorithm with accuracy closer to Sherlock but runtime closer to SCORE.

Rather than simply developing yet another localization algorithm with its own tradeoffs, we first develop a framework to understand the design space and answer the basic question: When is a given fault localization approach better and why? We observe that existing fault localization algorithms can be anatomized into three parts that correspond to how they *i*) model the system, *ii*) compute the likelihood of a component failure, and *iii*) explore the state space of potential failures. Delineating the choices made by an algorithm for each part enables systematic analysis of the algorithm's behavior.

Our anatomization also explains phenomena found empirically, but not fully explained, in existing work. For example, Kompella et al. [18] discover that noise leads SCORE to produce many false positives; they then suggest mitigation through additional heuristics. By contrast, we show that certain design choices of SCORE are inherently sensitive to noise, and changing these would lead to more robust fault localization than the suggested heuristic. As a second example, Pinpoint was found to have poor accuracy for simultaneous failures [8]. We show that this problem is fundamentally caused by how Pinpoint explores the state space of failures.

We use our understanding to devise a new fault localization algorithm, called Gestalt. Gestalt combines the best features of existing algorithms to work well in many networks and conditions. While Gestalt benefits from reusing existing components, we also introduce a new method for exploring the space of potential failures. Our method navigates a continuum between the extremes of greedy failure hypothesis exploration (e.g., SCORE) and combinatorial exploration (e.g., Sherlock).

Experiments on three real, diverse networks show that Gestalt simultaneously provides high localization accuracy and low computational cost. For instance, in Figure 1, we can see that Gestalt has higher accuracy than SCORE and Pinpoint; its accuracy is similar to Sherlock, but its running time is an order of magnitude lower.

In summary, this paper contributes a new fault localization algorithm that simultaneously provides high localization accuracy and low running time for a range of networks. Its design is not driven by our intuition alone, but by anatomization of the design space of fault localization algorithms. and by analysis of the ability of the design choices of existing algorithms to handle various characteristics of real networks (e.g., noise).Our analysis framework also explains why certain algorithms work well for some networks and not for others.

2. Related Work

Network diagnosis can be thought of as having two phases. The first processes available information (e.g., log files, passive or active measurements) to estimate system operation and is often used to *detect* faults. Several system-specific techniques exist for this phase [5, 9–11, 15, 19–21, 23–25]. Its output is often fed to a second phase that *localizes* faults. Localization identifies which system components are likely to blame for failing transactions.

Fault localization techniques are extremely valuable because information on component health may not be easily available in large networks and manual localization can lead to several hours of downtime. Even where component health information is available, it may be incorrect (as in the case of "gray failures" in which a failed component appears functional to liveness probes) or insufficient towards identifying culprits for failing transactions [6]. Fault localization has also been studied widely [3,6,8,11,13,14,16–18,26,27]. We focus on this second phase and ask: *given information from the first phase, which fault localization algorithm gives the best accuracy with the lowest overhead, and why?*.

Some diagnostic tools like [21, 23, 24] leave fault localization to a knowledgeable network operator and aim to provide the operator with a reduced dependency graph for a particular failure. While this is different from what we call fault localization in this paper, the automated fault-localization techniques we discuss can be used in those tools as well, to narrow down the list of suspects.

Steinder and Sethi [28] suvey the fault localization landscape but consider each approach separately. To the

impact of both precision and recall. It will be high if components deemed more likely are not actual failures (poor precision) or if actual failures are deemed unlikely (poor recall).

best of our knowledge, ours is the first work to analyze the design space for fault localization, and to use this insight to propose a better fault localization tool Gestalt.

3. Fault Localization Anatomy

We consider the following common fault localization scenario. The network is composed of components such as routers and servers. The success of a transaction in the network depends on the health of the components it uses. The goal of fault localization is to identify components that are likely responsible for failing transactions. While we use the term transaction for simplicity in this paper, it can be any indicator of network health (e.g., link load) for which we want to find the culprit component.

More formally, the state of the network is represented by a vector I with one element $I[j]$ per network component that represents the health of component j. Let O be a vector of observation data such that $O[k]$ represents whether transaction k succeeded. For example, O could represent the results of pings between different sources and destinations. The broad goal is to infer likely values of I that explain the observations O. Specifically, the fault localization algorithm outputs a sequence of possible state vectors $I_1, I_2, ..$ ordered in terms of likelihood.

We measure the goodness of an algorithm by its *diagnostic rank*: given ground truth about the components that failed denoted by I_{true}, the diagnostic rank is j if $I_{true} = I_j$ for some j in the output sequence; and n, the total number of possible state vectors, otherwise. For example, a network with two routers R and S and one link E between them will have a 3 element state vector denoting the states of R, S, and E respectively. Let us say that only router R has failed so $I_{true} = (F,U,U)$ where F denotes failed and U denotes up. If the output of the fault localization algorithm is $(U,F,U),(F,U,U),(U,U,F)$ then the diagnostic rank on this instance of running fault localization is 2 because one other component failure (router S) has been considered more likely. Lower diagnostic rank implies fewer "false leads" that an operator must investigate. A second metric for an algorithm is the computation time required to produce the ranked list given the observation vector O.

We find that practical fault localization algorithms can be anatomized into three parts: a system model, a scoring function, and a state-space explorer. First, any fault localization algorithm needs information such as which components are used by each transaction, and possible failure correlations between component failures (e.g., a group of links in a load-balancing relationship). Thus, localization algorithms start with a **system model** S that predicts the observations produced when the system is in state I. System models in past work are often cast in the form of a *dependency graph* between transactions and components but there is considerable variety in the types of dependency graphs used (§4.1).

Second, in theory fault localization can be cast as a Bayesian inference problem. Given observation O, rank system states I based on $P_S(I|O)$, the probability that I led to O when passed through the system model S. However, even approximate Bayesian inference [12, 22] can seldom handle the complexity of large networks [13]. So practical algorithms use a heuristic scoring function *Score* that maps each component to a metric that represents the likelihood of that component failing. The underlying assumption is that for two system states I_i and I_j and respective observations O_i and O_j predicted by S: $P_S(I_i|O) \geq P_S(I_j|O)$ when $Score(O_i,O) \geq Score(O_j,O)$, where O is the actual observation vector. This **scoring function** is the second part of the pattern.

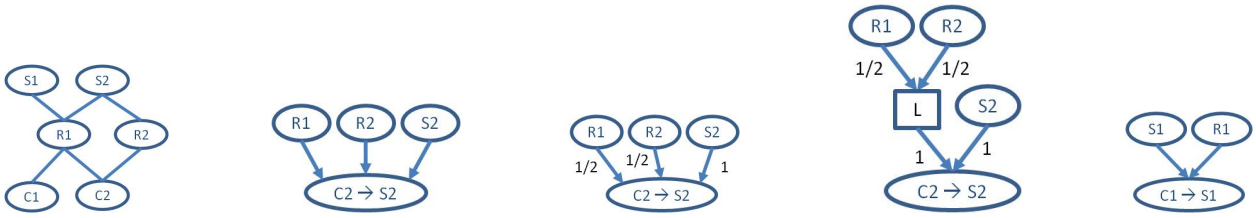
Finally, given the system model and scoring function the final job of a fault localization algorithm is to list and evaluate states that are more likely to produce the given observation vector. But system states can be exponential in the number of components since any combination of components can fail. Thus, localization algorithms have a third part that we call **state space exploration** in which heuristic algorithms are used to explore system states, balancing computation time with accuracy.

We do not claim that this pattern fits all possible fault localization algorithms. It does not fit algorithms based on belief propagation [26, 27]; such algorithms are computationally expensive and have not been shown to work with real systems. However, as Table 3 shows, this pattern does capture algorithms that have been evaluated for real networks, despite considerable diversity in this set.

4. Design Choices for Localization

We map existing algorithms into the three-part pattern by describing the choices they make for each part. §4.1-4.3 describes the choices, and §4.4 provides the mapping.

Prior algorithms also use different representations such as binary [8, 17, 18] or probabilities [14]) for transaction and component states. We use the 3-value representation from Sherlock [6] as it can model all prior representations. Specifically, the state of a component or transaction is a 3-tuple, $(p^{up}, p^{troubled}, p^{down})$, where p^{up} is the probability of being healthy, p^{down} that of having failed, and $p^{troubled}$ that of experiencing partial failure; p^{up}+$p^{troubled}$+p^{down}=1. The state of a completely successful or failed transaction or component is (1,0,0) or (0,0,1); other tuples represent intermediate degrees of health. A monitoring engine determines the state of a transaction in a system specific way; for example, a transaction that completes but takes a long time may be assigned $p^{troubled} > 0$.



(a) Network (b) DTL: $C_2 \rightarrow S_2$ (c) PTL: $C_2 \rightarrow S_2$ (d) PML: $C_2 \rightarrow S_2$ (e) $C_1 \rightarrow S_1$

Figure 2: An example network and models for two transactions.

	Failed component(s)			
	R1	R2	S2	R1&R2
DTL	0	0	0	0
PTL	1/2	1/2	0	3/4
PML	1/2	1/2	0	0

Table 1: Transaction state (p^{up}) predicted by different models for transaction $C_2 \rightarrow S_2$ in Figure 2

4.1 System Model

A system model represents the impact of network components on transactions. It can be encoded as a directed graph, where an edge from A to B implies that A impacts B. Three types of system models have been used by prior localization algorithms:

1. Deterministic Two Level (DTL) is a two-level model in which the top level corresponds to system components and the bottom level to transactions. Components connect to transactions they impact. The model assumes that components independently impact dependent transactions, and a transaction fails if any of its parent components fails.

2. Probabilistic Two Level (PTL) is similar to DTL except that the impact is modeled as probabilistic. Component failure leads to transaction failure with some probability.

3. Probabilistic Multi Level (PML) can have more than two levels; intermediate levels help encode more complex relationships between components and transactions such as load balancing and failover.

The network in Figure 2(a) helps illustrate the three models. The network has two clients (C_1, C_2), two servers (S_1, S_2), two routers (R_1, R_2), and several links. Transactions are requests from a client to a server ($C_i \rightarrow S_j$). Each request uses the shortest path, based on hop count, between the client and server. Where multiple shortest paths are present, as for $C_2 \rightarrow S_2$, requests are load balanced across those paths.

Assume that the components of interest for diagnosis are the two routers and the two servers. Then, Figures 2(b)-(d) show the models for the transaction $C_2 \rightarrow S_2$. Different models predict different relationships between the failures of components and that of the transaction. These predictions are shown in Table 1. For simplicity, the table shows the value of p^{up}; $p^{down} = 1 - p^{up}$ and $p^{troubled} = 0$ in this example. DTL predicts that the transaction fails when any of the components upon which it relies fails. Thus, the transaction is (incorrectly) predicted as always failing even when only one of the routers fails. PTL provides a better approximation in that the transaction is not deemed to completely fail when only one of the router fails. However, it still does not correctly model the impact of both routers failing simultaneously. PML is able to correctly encode complex relationships. While this example shows how PML correctly captures load balancing, it can also model other relationships such as failover [6]. However, this higher modeling fidelity does not come for free; as we discuss later, PML models have higher computational overhead.

In this network, the three models for the other three types of transactions ($C_1 \rightarrow S_{\{1,2\}}, C_2 \rightarrow S_1$) are equivalent. The model for $C_1 \rightarrow S_1$ is shown in Figure 2(e)

4.2 Scoring function

Scoring functions evaluate how well the observation vector predicted by the system model for a system state matches the actual observation vector. Let $(p^{up}, p^{troubled}, p^{down})$ be the state of a transaction in the predicted observation vector. Let $(q^{up}, q^{troubled}, q^{down})$ be the actual state determined by the monitoring engine. Then, the computation of various scoring functions can be compactly explained using the following quantities:

Explained failure	$eF = p^{down} q^{down}$
Unexplained failure	$nF = (1 - p^{down}) q^{down}$
Explained success	$eS = p^{up} q^{up} + p^{troubled} q^{troubled}$
Unexplained success	$nS = (1 - p^{up}) q^{up} + (1 - p^{troubled}) q^{troubled}$

eF is the extent to which the prediction explains the actual failure of the transaction, and nF measures the extent to which it does not. eS and nS have similar interpretations for successful transactions. We also define another quantity $TF = \Sigma(eF + nF)$, where the summation is over all elements of observation vectors. Because $eF + nF = q^{down}$, TF is the total number of failures in the actual observation vector.

	Failed component		
	R1	R2	S1
ΣeF	3	1	2
ΣnF	0	2	1
ΣeS	0	0	1
ΣnS	1	1	0
FailureOnly (ΣeF/TF)	1	1/3	2/3
InBetween (ΣeF/(TF+ ΣnS))	3/4	1/4	2/3
FailureSuccess (ΣeF+ΣeS)	3	1	3

Table 2: Score computed by different scoring functions for three possible failures.

Different scoring functions aggregate these basic quantities across observation elements in different ways. We find three classes of scoring functions:

1. FailureOnly (eF, TF): Such scoring functions only measure the extent to which a hypothesis explains actual failures. They thus use only eF and TF.

2. InBetween (eF, nS, TF): Such scoring functions only measure the extent to which a hypothesis explains failures and unexplained successes.

3. FailureSuccess (eF, eS): Such scoring functions measure *both* the extent to which a hypothesis explains failures and how well it explains successes.

Concrete instances of these classes are shown in Table 3. As expected, the score increases as eF and eS increase, and decreases when nF and nS increase. Given the large number of elements, each aggregates them in a way that is practical for high-dimensional spaces [4, 7].

Instead of analyzing every instance, in this paper we use a representative for each of the three classes. We find that the performance of different functions in a class is qualitatively similar. Our experiments use as representatives the functions used by SCORE (FailureOnly), Pinpoint (InBetween), and Sherlock (FailureSuccess).

To understand how different scoring functions can lead to different diagnoses, consider Figure 2 again. Assume that R_1 has failed and the actual state of four transactions is available to us. Two of these are $C_1 \rightarrow S_1$, both of which have failed (since they depend on $R1$); and the other two are $C_2 \rightarrow S_2$, one of which has failed (because it used $R1$, while the other used $R2$). Table 2 shows how the scoring functions evaluate three system states in which exactly one of R1, R2, and S1 has failed. The computation uses DTL for the system model. The top four rows show the values of the basic quantities. As an example, ΣeF is 3 in Column 1 because R1's failure correctly explains the three failed transactions; it is 1 in Column 2 because R2's failure explains the failure of only one transaction ($C_2 \rightarrow S_2$) and not of the two $C_1 \rightarrow S_1$ transactions.

The bottom three rows of the table show the scores of the three scoring functions for each failure. Even in this simple example, different scoring functions deem different failures as more or less likely. FailureOnly and InBetween deem $R1$ as the most likely failure that explains the observed data, FailureSuccess deems (incorrectly) that the data can be just as well be explained by the failure of $S1$. While it may appear that FailureSuccess is a poor choice, we show later that FailureSuccess actually works well in a variety of real networks.

4.3 State space exploration

State space exploration determines how the large space of possible system states (i.e., combinations of failed components) is explored. Prior work uses four types of explorers.

1. Independent explores only system states with exactly one component failure.

2. Joint$_k$ explores system states with at most k failures. It is a generalization of Independent (which is Joint$_1$).

3. Greedy set cover (Gsc) is an iterative method. In each iteration, a single component failure that explains the most failed transactions is chosen. Iterations repeat until all failed transactions are explained. Thus, it greedily computes the set of component failures that cover all failed transactions.

4. Hierarchical is also an iterative method. As in Gsc, in each iteration the component C that best explains the actual observations is chosen. However, a major difference is that if there are additional observations that C impacts, then these are added to the list of unexplained failures even if they were originally not marked as having failed in the input. Thus unlike Gsc, the set of unexplained failures need not decrease monotonically.

4.4 Mapping fault localization algorithms

Table 3 maps the fault localization portion of nine prior tools to our framework. Readers familiar with a tool may not immediately see how its computation maps to the choices shown because the original description uses different terminology. But in each case we have analytically and empirically verified the correctness of the mapping: composing the choices shown for the three parts leads to a matching computation (except for aspects mentioned below). Due to space constraints, we omit these verification experiments.

The last column lists aspects of the tool that are not captured in our framework. Most relate to pre- or post-processing data, e.g., candidate pre-selection removes irrelevant components at the start. The table does not list other suggestions by tool authors such as using priors that capture baseline component failure probabilities.

While the aspects we do not model are useful enhancements, they are complementary to the core localization algorithm. Our goal is to understand the behavior of fundamental choices made in the core algorithm. By

Tool	Target system	System Model	Scoring Function	State Space Exploration	Aspects not captured
Codebook [16]	Satellite comm. network	DTL,PTL	FailureSuccess ($\Sigma(eF+eS)$)	Independent	Codebook selection
MaxCoverage [18]	ISP backbone	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate post-selection, Hypothesis selection
NetDiagnoser [11]	Intra-AS, multi-AS internetwork	DTL	FailureOnly ($\frac{\Sigma eF}{TF}$)	Gsc	Candidate pre-selection
NetMedic [14]	Small enterprise network	PTL	FailureOnly (ΣeF)	Independent	Re-ranking
Pinpoint [8]	Internet services	DTL	InBetween ($\frac{\Sigma eF}{TF+\Sigma nS}$)	Hierarchical	
SCORE [17]	ISP backbone	DTL	FailureOnly($\frac{\Sigma eF}{TF}$)	Gsc	Threshold based hypothesis selection
Sherlock [6]	Large enterprise network	PML	FailureSuccess ($\prod(eF+eS)$)	$Joint_3$	Statistical significance test
Shrink [13]	IP network	PTL	FailureSuccess ($\prod(eF+eS)$)	$Joint_3$	
WebProfiler [3]	Web applications	DTL	InBetween ($\frac{\Sigma eF}{\Sigma nS+\Sigma eF}$)	$Joint_2$	Re-ranking

Table 3: Different fault localization algorithms mapped to our framework.

employing these choices, tools inherit their implications (§7) even when they use additional enhancements. Our paper abuses notation for simplicity; when we refer to a particular tool by name, we are referring to the computation that results from combining its three-part choices.

5. Network Characteristics

Localization algorithms must handle network characteristics that confound inference. We selected six such characteristics by simply asking: "what could go wrong with inference?" Clearly, dependency graph information can be incorrect (which we call uncertainty) and measurements may be wrongly recorded (which we call noise). We, and other researchers, have seen each characteristic empirically: e.g., noise in Lync and uncertainty in Exchange. We make no claim that our six characteristics are exhaustive but only that they helped explain why inference in the real networks we studied was hard.

In detail, the six characteristics we study are:

1. Uncertainty Most networks have significant non-determinism that makes the impact of a component failure on a transaction uncertain. For example, if a DNS translation is cached, a ping need not consult the DNS server; thus the DNS server failure does not impact the ping transaction if the entry is cached, but otherwise it does. This creates an uncertain dependency because the localization algorithm is not privy to DNS cache state. Load balancing is another common source of non-determinism e.g., $C_2 \rightarrow S_2$ transaction in Figure 2.

More precisely, if a component potentially (but not always) impacts a transaction failure, we call the dependency *uncertain*. A network whose system model contains uncertain edges is said to exhibit uncertainty. The degree of uncertainty is measured by the number of uncertain dependencies and the uncertainty of each dependency. Probabilistic models like PTL and PML can encode uncertainty while deterministic models cannot.

2. Observation noise So far, we assumed that observations are measured correctly. However, in practice, pings could be received correctly but lost during transmission to the stored log: thus an "up" transaction can be incorrectly marked as "down". Errors can also occur in reverse. In Lync, for example, the monitoring system measures properties of received voice call data to determine that a voice call is working; however, the voice call may still have been unacceptable to the humans involved. Both problems have been encountered in real networks [3, 11, 17, 18]. They can be viewed as introducing noise in the observation data that can lead sensitive localization algorithms astray. A network with 10% noise can be thought of as flipping 10% of the transaction states before presentation to the localization algorithm.

3. Covering relationships In some systems, when a particular component is used by a transaction, other components are used as well. For example, when a link participates in an end-to-end path, so do the two routers on either end. More precisely, component C covers component D if the set of transactions that C impacts is a superset of the transactions that D impacts.

Covering relationships confuse fault localization because any failed transaction explained by the covered component (link) can also be explained by the covering component (router). Other observations can be used to differentiate such failures; when a router fails, there may be path failures that do not involve the covered link. But some fault localization methods are better than others at making this distinction.

4. Simultaneous failures Diagnosing multiple, simultaneous failures is a well-known hurdle. Investigating k simultaneous failures among n components potentially requires examining $O(n^k)$ combinations of components. For example, in Lync, even if we limit localization to components that are actively involved in current transactions, the number of components can be around 600; naively considering 3 simultaneous failures as in $Joint_3$ can take days to run. The key characteristic is the maximum number s of simultaneous failures; the operator must feel that more than s simultaneous failures are extremely unlikely in practice.

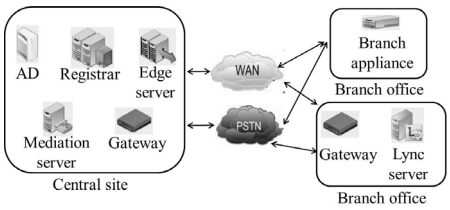


Figure 3: Lync architecture.

5. Collective impact So far, we assumed that a *single* component failure affects a transaction in possibly uncertain fashion. However, many networks exhibit a more complex dependency between a transaction and a *set* of components; the transaction's success depends on the collective health of the components in the set. For instance, when two servers are in a failover configuration, the transaction fails only if they both fail; otherwise, it succeeds. Collective impact is not limited to failover and load-balancing servers. Routers or links on the primary and backup paths in an IP network also have collective impact on message delivery. Multi-level models (e.g., PML) can model collective impact using additional logical nodes, but two-level models do not.[2]

6. Scale Network size impacts the speed of fault localization, which is key to fast recovery and high availability. Scale can be captured using the total number of components in the network and/or the typical number of observations fed to the localization algorithm. For Lync, the two numbers are 8000 and 2500.

6. Analysis methodology

Our goal is to analyze the relative merits of the choices made by various localization algorithms in the face of the network characteristics above. We do this by combining first principles reasoning and simulations of three diverse, real networks. This section describes our simulation method and the networks we study, and the next section presents our findings.

6.1 Simulation harness

In each simulation, we first select which system components fail. We then generate enough transactions—some of which fail due to the failed components—such that diagnosis is not limited by a lack of observations, as is true of large, busy networks [18, 21]. Finally, we feed these observations to the fault localization algorithm and obtain its output as a ranked list of likely failures.

[2]Our notion of collective impact differs from so called "correlated failures" in the literature which refers to components likely to fail together such as two servers are connected to the same power source.

Unless otherwise specified, the components to fail and the transaction endpoints are selected randomly. In practice, failures may not be random; we have verified that results are qualitatively similar for skewed failure distributions. In §9, we show that our findings agree with diagnosing real failures in Lync.

As is common, we quantify localization performance using *diagnostic rank* and *computation time*. Since diagnostic rank is the rank of components that have actually failed, it reflects the overhead of identifying real failures, assuming that operators investigate component failures in the order listed by the localization algorithm. Our simulation harness takes as input any network, any failure model, and any combination of localization methods.

6.2 Networks considered

To ensure that our findings are general, we study three real networks that are highly diverse in terms of their size, services offered, and network characteristics. The first network, Lync, supports interactive, peer-to-peer communication between users; the second, Exchange, uses a client-server communication model; and the third, Abilene, is an IP-based backbone. Each network has one or more challenging characteristics. For instance, Lync has significant noise and simultaneous failures while Exchange has significant uncertainty. To our knowledge we are the first to consider diagnosis in a Lync-like network.

1. Lync Lync is an enterprise communication system. that supports several communication modes, including instant messages, voice, video and conferencing. We focus on the peer-to-peer communication aspects of Lync. The main components of a Lync network are shown in Figure 3. Internal users are registered with registrars and authenticated with AD (active directory). Audio calls connect via mediation servers, and out of the enterprise into a PSTN (public switched telephone network) using gateway. Edge servers handle external calls. Branch offices connect to the main sites by a WAN and PSTN.

The deployment of Lync that we study spans many offices worldwide of a large enterprise. It has over 8K components and serves 22K users. We have information on the network topology and locations of users. For a two-month period, we also have information on failures from the network's trouble ticket database and on transactions from its monitoring engine.

2. Exchange Exchange is a popular email system. Its transactions include sending and receiving email and are based on client-server communication. Important components of an Exchange network include mail servers, DNS, and Active Directory(AD) servers.

We study the Exchange deployment used in [9], with 530 users across 5 regions. The network has 118 components. The number of hubs, mailboxes, DNS and AD servers in a region are proportional to the number of

users. AD servers are in a load balancing cluster; hubs, DNS and mailbox servers are in a failover configuration.
3. Abilene Abilene is an IP-based backbone that connects many academic institutions in the USA. The topology [1] that we use has 12 routers and 15 links, for a total of 27 network components. The workload used for Abilene consists of paths between randomly selected ingress and egress routers selected.

7. Analysis results

Table 4 summarizes our findings by qualitatively rating models, scoring functions and explorers on how they handle the six network characteristics. For each network characteristic (columns), it rates each method as good, OK, or poor. An empty (shaded) subcolumn for a characteristic implies that each row is qualitatively equivalent with respect to that characteristic. For instance, the choice of state space explorer has little impact on the ability to handle uncertainty. We focus on parts of the table where different options behave differently. Each such part highlights the relative merits of choices, and we use it later to guide the design of Gestalt.

7.1 Uncertainty

Uncertainty arises when the impact of a component on a transaction is not certain. Conventional wisdom is that deterministic models cannot handle uncertainty [6, 13, 14]. But we find that:

Finding 1 *In the presence of uncertainty, DTL suffices if the scoring function is FailureOnly.* Consider a DNS server D whose impact on a specific transaction, say ping 1, is uncertain. In DTL, this uncertainty must be resolved (since the model is binary) in favor of assuming impact; for instance, we must assume that ping 1 depends on the D even if it used a locally cached entry. (If we err in the opposite direction and assume that ping 1 does not depend on the D, we would never be able to implicate D if the cache is empty and D actually fails.)

If this assumption happens to be true, no harm is done. But if false (i.e., the transaction does not depend on the component), there are two concerns. First, consider the case when the the real failure was a different component; for example, ping 1 failed because some router R in the path failed and not because D failed. In that case, D may be considered a more likely cause of the failure of ping 1 than R; but this can increase the diagnostic rank of R by at most 1, which is insignificant.

The second, more important, concern is that the ability to diagnose the failure of the falsely connected component itself may be significantly diminished. For example, when D fails, other transactions, say ping 2 and ping 3, may succeed because they use cached entries. This can confuse the fault localization algorithm because it increases the number of unexplained successes nS attributed to D, and decreases eS, potentially increasing significantly the diagnostic rank of D.

FailureOnly functions are not hindered by the false connection because they use only eF and nF in computing their score. But FailureSuccess and InBetween are negatively impacted because they do use eS and nS.

Figure 4 provides empirical confirmation for this finding using Exchange which has significant uncertainty because of the use of DNS servers whose results can be cached. It plots the diagnostic rank for 1000 trials; in each trial, a single random failure is injected. Observe that DTL with FailureOnly handles uncertainty just as well as PML and PTL. By contrast, DTL with FailureSuccess has much worse diagnostic rank (50 versus 5 in some trials). An implication of Finding 1 is that if the network has only uncertainty, it can be best handled (with small computation time and comparable diagnostic rank) using DTL and FailureOnly.

7.2 Observation noise

Finding 2 *FailureSuccess is most robust to observation noise, followed by InBetween, and then by FailureOnly.* Intuitively, using more evidence and all available elements reduces sensitivity to noise. Noise turns successful transactions into apparent failures or vice versa. FailureOnly is the most impacted because it uses only failure elements. FailureSuccess is the least impacted as legitimate failures appearing as successes add to eS the same amount as that subtracted from eF, and vice versa.

Figure 6(a) confirms this behavior. We inject single failures in Abilene and introduce 0-50% noise. We run 100 trials for each noise level and plot the median diagnostic rank for each level. This graph uses DTL and Independent as the system model and state space explorer; the relative trends are similar with other combinations.

Finding 3 *Iterative state space explorers, Gsc and Hierarchical, are highly sensitive to noise.* This is because an erroneous inference (due to noise) made in an early iteration can cause future inferences to falter.

Figure 6(b) confirms this behavior. In this experiment, we introduced two independent failures in Abilene and 0-50% observation noise. The experiment uses DTL and FailureSuccess while varying the state space explorer; other combinations of model and scoring function produce similar trends. Figure 6(b) plots the median diagnostic rank across 100 trials. We see that Gsc and Hierarchical deteriorate with small amounts of noise. Finding 3 helps explain the extreme sensitivity of SCORE, which uses FailureOnly and Gsc, to noise, that prior work [18] empirically observed but did not fully explain.

7.3 Covering relationships

Recall that a component C covers a component D if

	Uncertainty	Observation Noise	Covering relationship	Simultaneous failures	Collective Impact	Scale
DTL	Good w/ FailureOnly. Poor w/ other scoring funcs.				Poor	Good
PTL	Good				Poor	OK
PML	Good				Good w/ Joint$_k$. Poor otherwise.	OK
FailureOnly(FO)	Good	Poor	Poor			Good
InBetween	Good w/ PTL, PML Poor with DTL	OK	Good			OK
FailureSuccess(FS)	Good w/ PTL, PML. Poor with DTL	Good	Good			OK
Independent(Ind)		Good		Poor	Poor	Good
Joint$_k$(Jt_k)		Good		Good (s≤k). Poor (s>k)	Good (c≤k). Poor (c>k)	Poor
Gsc		Poor		Good*	Poor	Good
Hierarchical		Poor		Poor	Poor	OK

Table 4: Effectiveness of diagnostic methods with respect to factors of interest. * depends on the network.

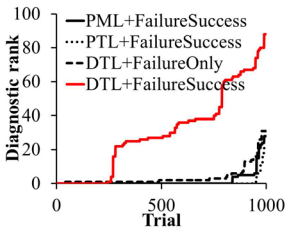


Figure 4: DTL can handle uncertainty when used with FailureOnly. [Exchange]

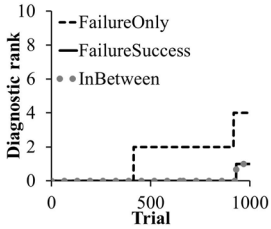
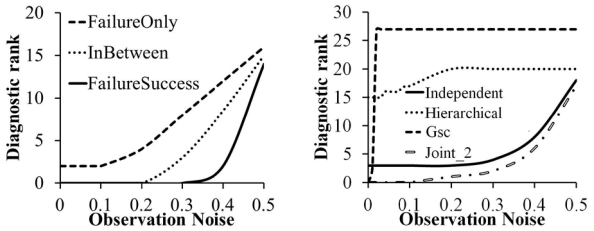


Figure 5: FailureOnly performs poorly for covering relationships. [Abilene]



(a) Scoring functions (b) State space explorers

Figure 6: Sensitivity to observation noise. [Abilene]

the set of transactions that D impacts is a subset of those that C impacts. In other words, when a transaction that D impacts fails, it is impossible to distinguish a failure of C from that of D by looking only at failures.

Finding 4 *For covering relationships, FailureOnly scoring functions should not be used.* Other scoring functions (FailureSuccess, InBetween) can better disambiguate the failures of the covering and covered component because they use successful transactions (eS, nS) as well, and not only failed ones. For instance, consider a failed link. All failed transactions due to the link can also be explained by the failure of the attached routers. By using successful transactions that include the routers but not the failed link, the scoring function can assign a higher likelihood to link failure than router failure.

Figure 5 verifies Finding 4 by showing the results of an experiment using Abilene, which has many covering relationships. We randomly introduced a single failure in the network and diagnosed it using different scoring functions (combined with DTL and Independent). We see that FailureOnly has the worst performance with non-zero diagnostic rank in 60% of the trials while the other two methods have rank 0 most of the time.

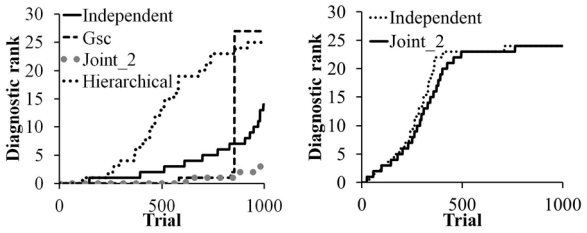
We note that FailureOnly has been used by several tools to diagnose ISP backbones [11, 17, 18], which have many covering relationships. Finding 4 suggests that the localization accuracy of these tools can be improved by changing their scoring function.

7.4 Simultaneous failures

We now discuss simultaneous failures of components that have *independent* impact on transactions. The next section discusses *collective* impact.

Finding 5 *For a small number of simultaneous failures (s≤k), Joint$_k$ is best and Hierarchical is worst.* The effectiveness of Joint$_k$ follows because it examines all system states with k or fewer failures. Hierarchical does poorly because its clustering approach forces it to explain more failures than needed. Suppose transactions O_1, O_2, O_3 have failed and component C explains O_1 and O_2 and no other component explains more failures. Suppose, however, that C also impacts transaction O_4. Then Hierarchical will add C to the cluster but will also add transaction O_4 as a new failed transaction to be explained by subsequent iterations. Intuitively, the onus of explaining more failures than those observed can lead Hierarchical astray.

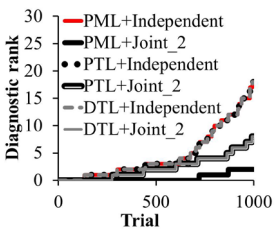
Figure 7(a) shows the performance of different state space explorers when diagnosing two (randomly picked) simultaneous failures in Abilene. The graph uses PML



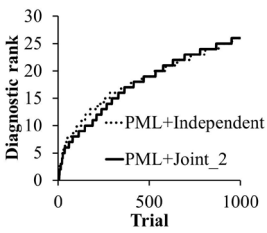
(a) 2 simultaneous failures

(b) 3-4 simultaneous failures

Figure 7: Ability of state space explorers to handle simultaneous failures. [Abilene]



(a) 2 collective failures



(b) 3 collective failures

Figure 8: Ability of a model+state space explorer to handle collective impact failures. [Abilene]

and FailureSuccess; other combinations produce similar trends. We see that Joint$_k$ is highly effective (rank 2 or less), and Hierarchical is poor (rank $>$ 20 in 25% of trials). Gsc has bimodal behavior with a rank $>$ 25 in a small fraction of trials. Closer investigation confirms that these trials involve the simultaneous failures of two components that together cover a third component.

Finding 5 explains why Pinpoint [8], which uses Hierarchical, has poor performance (see Figure 4 in [8]) for even two simultaneous failures, despite the handling of simultaneous failures being an explicit goal of Pinpoint. It suggests that replacing Hierarchical state space exploration in Pinpoint (with, say, *Joint$_2$*) while keeping the same system model and scoring function would improve Pinpoint's diagnosis of simultaneous failures.

7.5 Collective impact

We now study simultaneous failures of components that have a collective impact on transactions by being, for instance, in a load balancing or failover relationship. We find that in such cases, the choice of system model and state space explorer should be jointly made. We explored two cases: when the number s of failed components in a collection is small ($s \leq k$), and when it is large ($s > k$).

Finding 6 *For diagnosing a small number of simultaneous failures in a collection ($s \leq k$), combining PML and Joint$_k$ is most effective; any other system model or state space explorer leads to poor diagnosis.* This is because, among existing models, only PML can encode collective impact relationships. Other models represent approximations that can be far from reality. However, picking the right model is not enough. The state explorer must also consider simultaneous failure of these components. Among existing state space explorers, only Joint$_k$ has this property. Independent does not consider simultaneous failures, and Gsc and Hierarchical assume that components have independent impact.

Figure 8(a) demonstrates this behavior. We modeled failures among components with collective impact in Abilene as follows. Each trial randomly selects a pair of nodes that has two vertex-disjoint paths between them. For messages between these nodes, the two paths can be considered to be in a failover relationship with collective impact. We then introduced a randomly selected failure along each path. Thus, all messages sent between the pair of nodes will now fail. For 1000 such trials, the graph plots the diagnostic ranks of several combinations of system model and state space explorer. It uses FailureSuccess for scoring function, but others yield similar results. We omit results for Gsc and Hierarchical; they had worse performance than Independent. As we can see, only PML+Joint$_2$ is effective.

This result implies that half-way measures are insufficient for diagnosing collective impact failures. We must both model relationships (PML) and explore joint failures (Joint$_k$). Localization suffers if either choice is wrong. For example, Shrink [13] uses PTL with Joint$_k$ even though it targets IP networks which may have potentially many failover paths. Finding 6 suggests that Shrink would do better to replace PTL with PML.

8. Gestalt

The insights from the analysis above led us to develop Gestalt. It combines ideas from existing algorithms and also includes a new state space exploration method.

For the system model, Gestalt uses a hybrid between DTL and PML that combines the simplicity of DTL (fixed number of levels, deterministic edges) with the expressiveness of PML (ability to capture complex component relationships). Our model has three levels, where the top level corresponds to system components that can fail independently and the bottom level to transactions. An intermediate level captures collective impact of system components. Instead of encoding probabilistic impact on the edges, the intermediate node encodes the *function* that captures the nature of the collective impact. The domain of this function is the combinations of states of the parent nodes, and the range is the impact of each combination on the transaction. Figure 9(a) shows how Gestalt models the example in Figure 2a. The intermediate node I encodes the collective impact of R_1 and R_2.

Algorithm 1: Pseudocode for Gestalt

```
1: H_all = {};
2: For each hitRatio in 1, 0.95, ··· 0 do
3:     H_curr = (); //current hypothesis
4:     O_unexp = O_all; //unexplained observations
5:     H_all += GenHyp(I, O_unexp, hitRatio, H_curr);
6: Return H_all

   GenHyp(i, O_unexp, hitRatio, H_curr)
1: H_return = {H_curr};
2: C_new = NewCandidates(hitRatio, O_unexp);
3: For each c in C_new
4:     hyp_new = (hyp, c);
5:     If (i == k)
6:         H_return += hyp_new;
7:     Else
8:         O_exp = ExpObs(hyp_new, O_unexp);
9:         H_return += GenHyp(i+1, O_unexp − O_exp, hitRatio,
   hyp_new);
10: Return H_return

   NewCandidates(hitRatio, O_unexp)
1: C_new = {};
2: For each c in CandidatePool
3:     If (HitRatio(c) ≥ hitRatio)
4:         C_new += c;
5: score_max = MaxScore(C_new, O_unexp);
6: score_noise = Noise_thresh × |O_unexp|;
7: For each c in C_new
8:     If (Score(c) < score_max − score_noise)
9:         C_new −= c;
10: Return C_new
```

The function represented by I is shown in the figure, which shows values only for p_{up} (p_{down}=1−p_{up}).

While for this example, PML too has only three levels, Figure 9(b) illustrates the difference between PML and Gestalt. Here, to reach S, C spreads packets across $R1$ and $R2$, and $R2$ spreads across $R3$ and $R4$. Figures 9(c) and 9(d) show PML and Gestalt models for this network.

Another difference between PML and our model is how we capture single components with uncertain impact on a transaction (e.g., a DNS server whose responses may be cached). Gestalt models these with 3 levels too. An intermediate node captures the uncertainty from the component's state to its impact on the transaction. It may deem, for instance, that the transaction will succeed with some probability even if the component fails.

As scoring function, we use FailureSuccess because of its robustness to noise and covering relationships (Findings 2 and 3). By explicitly modeling uncertainty (unlike DTL), the combination of our model and FailureSucess is robust to uncertainty as well (Finding 1).

For state space exploration, we develop a method that has the localization accuracy of Joint$_k$ and the low computational overhead of Gsc. It is based on the following observations. Gsc is susceptible to covering relationships because many failure combinations can explain the observations and Gsc explores only a subset, ignoring others (Finding 5). Gsc is susceptible to noise because noise can make it pick a poor candidate and rule out other possibilities (Finding 3). The diagnostic accuracy of Joint$_k$ for collective impact failures stems from the fact that it explores combinations of at most k failures; exploring a smaller number does not help (Finding 6). But because its exploration is fully combinatorial, it has a high computational overhead.

Our new exploration method is shown in Algorithm 1. It takes two parameters as input. The first is $Noise_{thresh}$, the percentage of observation noise expected in the network, which can be estimated from historical data. Given ground truth (post resolution) about a failure and the transaction logs, the percentage of transactions that cannot be explained by the ground truth reflects the level of observation noise. In Lync, we found this to be around 10%. The second parameter is k, the maximum number of simultaneous failures expected in the network. It can also be gleaned from historical failure data.

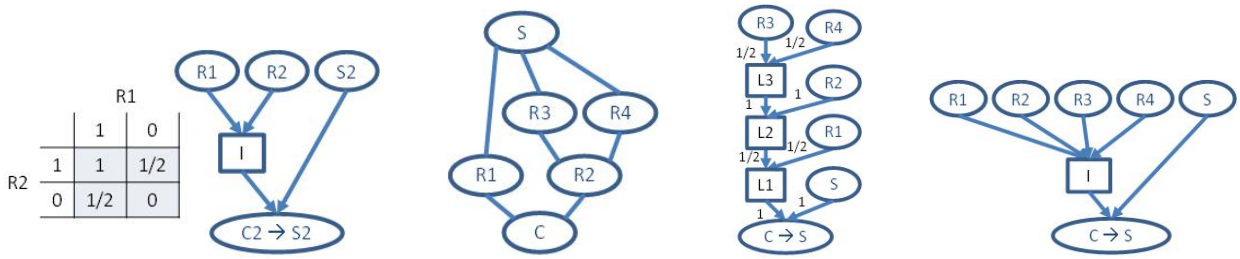
The candidate failures that we explore are single component failures and combinations of up to k components with collective impact. This candidate pool explicitly accounts for collective impact failures (making them diagnosable, unlike in Gsc). It is also much smaller than the pool considered by Joint$_k$ which includes all possible combinations of up to k failures. The output of the exploration is a ranked list of hypotheses, where each hypothesis is a set of at most k candidates from the pool.

These sets are computed separately for different thresholds of hit ratio [17]. The hit ratio of a candidate is the ratio of number of failed versus total transactions in which the component(s) participated. Iterating over candidates in decreasing order of hit ratios gives us a systematic way of exploring failures while focusing on more likely failures first because actual failures are likely to have larger hit ratios. Hit ratios are not used in the scoring function.

For a given hit ratio threshold, the hypothesis sets are built iteratively (i.e., not all possible sets are considered) in k steps. We start with the empty set. At each step, each set is forked into a number of child sets, where each child set has one additional candidate than the parent set.

The child candidates are computed as follows. Let O_{unexp} be the set of observations whose status cannot be explained by the parent set (i.e., the status does not match what would be predicted by the system model). Initially, when the parent set is empty, this set equals O_{all}, the set of all observations. Then, we first compute the score of each candidate in the entire pool with hit ratio higher than the current threshold. This computation uses the scoring function (FailureSuccess) and is done with respect to O_{unexp}. Candidates more likely to explain the as yet unexplained observations will have higher scores.

If there were no observation noise, candidates with the maximum score can be used as child candidates because they best explain the remaining unexplained ob-



(a) Gestalt model for Figure 2a

(b) Another example network

(c) PML for Fig. 9(b)

(d) Gestalt model for Figure 9(b)

Figure 9: Modeling in Gestalt

servations. But due to noisy observations, the score of actual failures may go down and the score of some other candidates may go up. By focusing only on candidates with the maximum score, we run the risk of excluding actual failures from the set, like Gsc.

We thus cast a wider net; the width of the net is proportional to expected noise. The score of the actual culprit can be expected to reduce due to observation noise by $score_{noise} = Noise_{thresh} \times |O_{unexp}|$. The selected candidates are those with scores higher than $score_{max} - score_{noise}$, where $score_{max}$ is the maximum score across all candidates. This reduces chances of missing actual failures. $Noise_{thresh}$ and k enable Gestalt to explore the continuum between Gsc and exhaustive search. $Noise_{thresh}$ set to 0 mimics Gsc (but handles covering relationship), and $Noise_{thresh}$ set to 100 mimics Joint$_k$.

9. Gestalt Evaluation

We now evaluate Gestalt and compare it to 3 existing algorithms that use very different techniques. We start with Lync and use the algorithms to diagnose real failures using real transactions available in system logs. Based on information from days prior to the failures we diagnose, we set $Noise_{thresh}$=10% and k=2 for Gestalt.

	Original recovery delay (days, hh:mm)	# potential failed comps	Gestalt diagnostic rank	Gestalt run time (mm:ss)
1	0,01:50	196	11	4:02
2	0,00:50	625	7	2:59
3	0,01:55	552	6	0:05
4	0,22:05	608	9	0:05
5	1,23:45	521	7	0:12
6	0,10:55	655	6	0:21
7	14,06:25	676	12	2:43
8	0,01:45	571	13	1:06
9	0,20:15	562	13	0:23
10	0,08:20	455	3	1:03

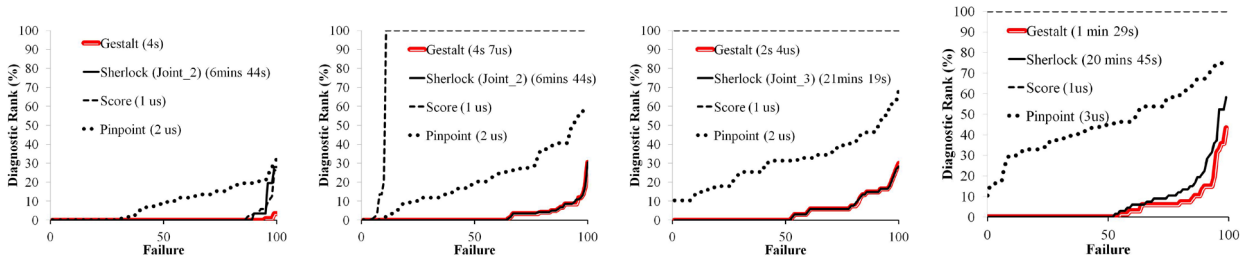
Table 5: Statistics for a sample of real failures in Lync.

Figure 1(a) shows the results for a number of failures seen in a two month period (the actual failure count is hidden for confidentiality). The legend shows the median running time for the algorithms on a 3 GHz dual-core PC. We see that SCORE and Pinpoint perform poorly. Gestalt and Sherlock perform similarly, but the running time of Gestalt is lower by more than an order of magnitude. This is despite the fact that we ran Sherlock with Joint$_2$. Using Joint$_3$, recommended in the original Sherlock paper [6], would have taken $\sim$ 20 hours per failure.

Table 5 provides more details for ten sample failures in the logs. We see that the time it took for the operators to manually diagnose these failures, reflected in the original recovery delay, was very high. The median time was around 8 hours, though it took more than a day for two failures. The primary reason for slow recovery time was the large diagnosis time due to the number of network components that had to be manually inspected[3]. The table lists the number of components involved in failing transactions as an estimate of the number of possible components that might need to be checked. Of course, using domain knowledge and expertise, an operator will only check a subset of these components; but the estimate underscores the challenge faced by operators today. We see that using Gestalt, the operator will have to check only 3-13 components before identifying the real culprits compared to 196-655 components for manual diagnosis, significantly reducing diagnosis time. The run time for Gestalt to whittle down the list of suspects by 1-2 orders of magnitude is at most a few minutes.

We next consider simulated failures in the Exchange network. Figures 10(a) and 10(b) show results for diagnosing one and two component simulated failures. We again used Joint$_2$ for Sherlock and k=2 and $Noise_{thresh}$=0 for Gestalt. As expected based on our earlier analysis, Score does very well for single failure scenarios, but suffers in two-failure scenarios due to covering relation-

[3]In Lync, once a problem was diagnosed, service was restored quickly by repair or diverting transactions around the failed component.



(a) Single failure (b) 2 simultaneous failures (c) 3 simultaneous failures (d) 4 simult. failures, 1% noise

Figure 10: Diagnostic efficacy of different algorithms with Exchange network with different number of failures.

ships. Sherlock and Gestalt do well for both cases, but Sherlock takes two orders of magnitude more time.

In order to experiment with more simultaneous failures and Joint_3, we reduced the size of the Exchange network by half (to 67 components). Figures 10(c) and 10(d) show the results for three failures and for four failures with 1% observation noise. In the latter case, we run Gestalt with $Noise_{thresh}$=1%. We see that Gestalt matches Sherlock's diagnostic accuracy for three failures, with running time that is two orders of magnitude faster. For four failures, Gestalt has better diagnostic accuracy than Sherlock because it accounts for noise. Its running time is still better by 20x, even though noise makes it explore more failure combinations.

We omit results for Abilene, but we found them to be qualitatively similar to those above. Gestalt had better diagnostic efficacy than SCORE and Pinpoint for all cases. Gestalt matched Sherlock's accuracy for most cases and exceeded it in the presence of noise and more than three simultaneous failures. Its running time was 1-2 orders of magnitude lower than Sherlock.

10. Conclusion

We presented Gestalt, a fault localization algorithm that borrows the best ideas from prior work and includes a new state space explorer that represents a continuum between greedy, low-accuracy exploration and combinatorial, high-overhead exploration. The result is an algorithm that simultaneously provides high localization accuracy and low overhead for a range of networks. Its design is guided by an analysis framework that anatomizes the design space of fault localization algorithms and explains how the design choices of existing algorithms interact with key characteristics of real networks. Beyond the specific algorithm it helped develop, we hope this framework takes a modest step towards understanding the *gestalt* of fault localization.

11. References

[1] Abilene Topology. `http://totem.run.montefiore.ulg.ac.be/files/examples/abilene/abilene.xml`, 2005.

[2] Microsoft Lync. http://en.wikipedia.org/wiki/Microsoft_Lync, 2012.

[3] AGARWAL, S., LIOGKAS, N., MOHAN, P., ET AL. Webprofiler: Cooperative diagnosis of web failures. In *COMSNET* (2010).

[4] AGGARWAL, C. C. Re-designing distance functions and distance-based applications for high dimensional data. In *SIGMOD Record* (2001).

[5] AGUILERA, M. K., MOGUL, J. C., WEINER, J. L., ET AL. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[6] BAHL, P., CHANDRA, R., GREENBERG, A., ET AL. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *SIGCOMM* (2007).

[7] BEYER, K., GOLDSTEIN, J., RAMAKRISHNAN, R., ET AL. When is "nearest neighbor" meaningful? In *ICDT* (1999).

[8] CHEN, M., KICIMAN, E., FRATKIN, E., ET AL. Pinpoint: Problem determination in large, dynamic, internet services. In *IPDS* (2002).

[9] CHEN, X., ZHANG, M., MAO, M., ET AL. Automating network application dependency discovery:experiences, limitations, and new solutions. In *OSDI* (2008).

[10] CUNHA, T., TEIXEIRA, R., FEAMSTER, N., ET AL. Measurement methods for fast and accurate blackhole identification with binary tomography. In *IMC* (2009).

[11] DHAMDHEREY, A., TEIXEIRA, R., DOVROLIS, C., ET AL. Netdiagnoser: Troubleshooting network unreachabilities using end-to-end probes and routing data. In *CoNEXT* (2007).

[12] HECKERMAN, D. A tractable inference algorithm for diagnosing multiple diseases. In *Uncertainty in Artificial Intelligence* (1989).

[13] KANDULA, S., KATABI, D., AND VASSEURI, J.-P. Shrink: A tool for failure diagnosis in ip networks. In *MineNet workshop* (2005).

[14] KANDULA, S., MAHAJAN, R., VERKAIK, P., ET AL. Detailed diagnosis in computer networks. In *SIGCOMM* (2009).

[15] KATZ-BASSETT, E., MADHYASHTA, H. V., JOHN, J. P., ET AL. Studying black holes in the internet with hubble. In *NSDI* (2008).

[16] KLINGER, S., YEMINI, S., YEMINI, Y., ET AL. A coding approach to event correlation. In *International Symposium on Integrated Network Management* (1995).

[17] KOMPELLA, R. R., YATES, J., GREENBERG, A., ET AL. IP fault localization via risk modeling. In *NSDI* (2005).

[18] KOMPELLA, R. R., YATES, J., GREENBERG, A., ET AL. Detection and localization of network blackholes. In *Infocom* (2007).

[19] LAKHINA, A., CROVELLA, M., AND DIOT, C. Diagnosing network-wide traffic anomalies. In *SIGCOMM* (2004).

[20] MAHAJAN, R., SPRING, N., WETHERALL, D., ET AL. User-level internet path diagnosis. In *SOSP* (2003).

[21] MAHIMKAR, A., GE, Z., SHAIKH, A., ET AL. Towards automated performance diagnosis in a large iptv network. In *SIGCOMM* (2009).

[22] MURPHY, K. P., WEISS, Y., AND JORDAN, M. I. Loopy belief-propagation for approximate inference: An empirical study. In *Uncertainty in Artificial Intelligence* (1999).

[23] NAGARAJ, K., KILLIAN, C., AND NEVILLE, J. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI* (2012).

[24] OLINER, A. J., AND AIKEN, A. Online detection of multi-component interactions in production systems. In *DSN* (2011).

[25] REYNOLDS, P., WEINER, J. L., MOGUL, J. C., ET AL. Performance debugging for distributed systems of black boxes. In *WWW* (2006).

[26] RISH, I. Distributed systems diagnosis using belief propagation. In *Allerton Conf. on Communication, Control and Computing* (2005).

[27] STEINDER, M., AND SETHI, A. Probabilistic fault localization in communication. In *IEEE/ACM Trans. Networking* (2004).

[28] STEINDER, M., AND SETHI, A. A survey of fault localization techniques in computer networks. In *Science of Computer Programming* (2004).