Group Communication: Helping or Obscuring Failure Diagnosis?

Soila Pertet, Rajeev Gandhi and Priya Narasimhan

CMU-PDL-06-107
June 2006

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Replicated client-server systems are often based on underlying group communication protocols that provide totally ordered, reliable delivery of messages. However, in the face of a performance fault (e.g, memory leak, packet loss) at a single node, group communication protocols can cause correlated performance degradations at non-faulty nodes. We explore the impact of performance-degradation faults on token-ring and quorum-based group communication protocols in replicated systems. By empirically evaluating these protocols, in the presence of a variety of injected faults, we investigate which metrics are the most/least appropriate for failure diagnosis. We show that group communication protocols can both help and obscure root-cause analysis, and present an approach for fingerpointing the faulty node by monitoring OS-level and protocol-level metrics. Our empirical evaluation suggests that the root-cause of the failure is either the node exhibiting the most anomalies in a given window of time or the node with an "odd-man-out" behavior, e.g., if a node displays a surge in context-switch rate while the other nodes display a dip in the same metric.

Keywords: Problem diagnosis, Fingerpointing, Group communication

1. Introduction

Distributed systems contain multiple components that can interact across multiple nodes in sometimes unforeseen and complicated ways. As a result, determining the root cause of failures in these systems can be frustrating, and might take several hours or even days. Delays in determining the root-cause of faults can adversely impact system availability. *Fingerpointing*, also known as root-cause analysis or failure diagnosis, involves detecting errors and/or failures and assigning blame to their underlying cause. Fingerpointing in distributed systems has mostly been studied in enterprise systems with databases that protect data in the case of failures. Our goal is to study fingerpointing in distributed, replicated systems with the aim of improving system availability and providing better fault containment.

Replication is a common technique used for providing fault-tolerance to distributed, client-server applications. The idea behind replication is to provide multiple copies, or replicas, of servers so that even if one replica crashes or fails, another replica can take over to continue the operation. This means that replicas must be consistent in state and behavior, i.e., they must be substitutable for each other even if they run on different nodes in the system. To accomplish this, replicated systems often exploit group communication protocols [7] as an underlying building-block for providing totally ordered, reliable message delivery of all messages.

Fingerpointing makes a difference in distributed, replicated systems because it can facilitate proactive and targeted fault-recovery. First, if the root-cause of a failure can be diagnosed quickly, this gives us a window of opportunity to initiate recovery proactively, instead of waiting for the failure to be detected through the timeout or membership mechanisms of the group communication protocols. Secondly, knowledge of the root-cause can assist in deciding the appropriate course of action for recovery; if the root-cause is a CPU overload on some node, load-balancing might be more appropriate from a system-availability viewpoint than rebooting/removing the node (as group communication protocols are prone to do). While fingerpointing is clearly useful in distributed, replicated systems, the use of group communication protocols in these systems can often aggravate the process of diagnosing the root-cause of failures. These protocols inevitably introduce some form of coupling between nodes in the system. In fact, as our experimental results show, this innate coupling can cause fault-manifestations to "travel" across the distributed system. As hidden dependencies and coupling in a distributed system increase, fingerpointing is likely to become harder.

In this paper, we focus on fingerpointing performance problems in distributed, replicated systems. We perform our investigations for client-server applications that use two different group communication protocols, namely, a token-ring protocol called Spread [3] and a quorum-based protocol called Castro-Liskov BFT [6]. We inject a variety of performance-degrading faults (such as a memory leak, process hangs, packet-loss and abrupt crashes) into our target systems. We investigate the effectiveness of a black-box fingerpointing approach, where our root-cause analysis is driven solely by the behavior of operating-system (OS) performance metrics such as CPU and memory usage. We determine what kinds of faults can and cannot be diagnosed using this black-box approach, and then discuss a white-box fingerpointing approach, where we can exploit group communication metrics for more effective failure diagnosis. The main contributions of this paper are:

Influence of group communication protocols on failure-masking and fingerpointing. We evaluate the behavior of two group communication protocols under various performance-degrading faults. The two protocols (Spread and BFT) were independently developed, are based on different algorithms (token-ring vs. quorum), and are implemented differently (daemon vs. library). We discuss how the underlying protocol influences failure-masking and fingerpointing.

Influence of group communication protocols on fingerpointing. We evaluate the behavior of two group communication protocols under various performance-degrading faults. The two protocols (Spread and BFT) were independently developed, are based on different algorithms (token-ring vs. quorum), and are implemented differently (daemon vs. library). We discuss how the underlying protocol influences fingerpointing, and how our fingerpointing approach complements and improves upon the protocols own failure-detection mechanisms.

This paper is organized as follows: Section 2 provides a brief description of Spread and BFT; Section 3 defines the problem and gives motivating examples for our fingerpointing approach; Section 4 presents our monitoring framework, fault-injection strategy and fingerpointing algorithm. Section 5 details our empirical evaluation, Section 6 discusses related work, and Section 7 outlines our conclusions.

2. Group Communication Protocols

Group communication systems [7] provide support for group membership and reliable, totally ordered message delivery. A node group consists of all of the nodes in the system, with each group receiving the same set of messages in the same system-wide (i.e., total) order. A replica group, layered on top of the node-group membership, is a virtual addressing mechanism that corresponds to a unique replicated server at the application level, with the group's members corresponding to the server's replicas. Node-group (replica-group) membership keeps track of which nodes (replicas) are in the group, and provides notifications when a node (replica) is added or removed. The application does not need to be concerned with the location or number of the replicas; the protocol handles these details transparently by maintaining membership information and routing messages appropriately.

Guaranteeing system-wide consistent views of group membership and providing reliable, totally ordered message-delivery requires consensus or agreement protocols. These can be expensive, and can also introduce coupling between nodes participating in the consensus algorithms. We investigate how this coupling influences failure-masking and fingerpointing in actively replicated client-server systems built on top of the Spread [3] and Castro-Liskov BFT [6] group communication protocols. Our observations are not an indictment against either protocol or against group communication in general; in fact, both Spread and BFT are well-engineered and effective at supporting active replication. Our focus instead is to investigate if and how these protocols impact the failure-masking and timing properties that we have come to expect of active replication.

We briefly describe token-ring and quorum-based group communication protocols, with particular focus on how Spread and BFT implement group membership and message retransmissions.

2.1. Token-Ring Protocols

Token-ring protocols impose a *logical* ring on the set of nodes constituting the node-group membership. A special message called the token circulates within the node-group, sequentially from one node to the next, using a point-to-point connection between each adjacent pair of nodes within the virtual ring. A node is allowed to broadcast messages to the other nodes only when it holds the token; after each node is done broadcasting its messages, it passes the token onto the next node in the ring. The token carries enough information to allow messages to be ordered as the token "visits" each node in turn; the token's circulation is critical to achieving consensus on the system-wide (i.e., ring-wide) membership and ordering of messages.

Spread [3] is a token-ring protocol that provides reliable, totally ordered message delivery, in the face of crash faults, communication faults and network partitions. On every node that is a member of the ring, there runs a Spread daemon that allows multiple local processes to obtain access to group communication services. A node is, thus, synonymous with its Spread daemon. In a fault-tolerant client-server application, each server replica and each client must be connected to a Spread daemon, although not necessarily to a unique one; *f+1* server replicas are required to tolerate *f* crash faults.

Group-membership changes can be triggered by a number of factors: a timeout expiring that indicates that a node might have crashed, a new message from a non-member, a request from a member to leave the group, etc. A group-membership change can take time, and can prevent the application from making progress until the membership process converges to install a new view of the ring's membership; thus, membership changes can be detrimental to real-time operation. Spread ensures that membership-changes converge by excluding, from the new ring's membership, any nodes that are suspected of failure during an ongoing membership-change. Once a ring is formed, each node in the ring keeps track of the sequence numbers of the messages that it receives. Spread uses the token to track the highest sequence number of all messages that have been broadcast within the ring, and handles message losses through retransmissions.

2.2. Quorum-Based Protocols

A quorum-based system for a replicated server is a collection of subsets (known as quorums) of the server replicas, where all pairs of subsets intersect. Each quorum can provide sufficient availability for the system, while operations on distinct quorums preserve consistency. Castro-Liskov BFT [6] is a quorum-based group communication protocol that tolerates crash, communication and Byzantine/arbitrary faults. Tolerating Byzantine faults requires a greater degree of replication, i.e., *3f+1* replicas to tolerate *f* faults. Each application process must be compiled with the BFT library, in order to access group communication services.

BFT uses quorums (containing *2f+1* replicas each) to totally order messages despite failures. During fault-free operation, the client uses a point-to-point connection to send a request to a designated leader replica. The leader then uses a 3-phase protocol to multicast the message to follower replicas, thereby enforcing a specific ordering of messages. Each replica processes the request according to the total order and sends a reply directly to the client. The client waits to receive *f+1* identical replies from the server replicas before processing them. If the client does not receive a reply within a specified amount of time, it re-broadcasts its request, but now to all server replicas. The follower replicas unicast this request to the primary and wait for a timeout to expire. If the timeout expires before a follower receives new requests for execution from the leader, the follower multicasts a view-change message (BFT view-changes correspond to Spread's node-group membership-changes). If this view change is valid, a new leader is elected. The new leader then initiates a 3-phase protocol for system-wide agreement on the set of pending messages to be delivered in the old view, and then installs the new view. Message losses are handled through retransmissions.

Although BFT tolerates Byzantine faults, our focus is on performance failures that can be compared across BFT and Spread. Thus, malicious faults are currently outside the scope of our experiments.

3. The Case for Fingerpointing

Group communication protocols use timeouts to detect failures, and attempt to reduce all failures to membership changes, e.g., a slow node, a lossy network all ultimately provoke a membership change. However, some faults can hide under the radar of the protocol's timeouts and cause lingering performance problems to exist and even propagate while within the system, as shown in the motivating examples below. In such cases, active replication's failure masking (and therefore, its support for real-time fault-tolerant operation) might be compromised. The addition of a fingerpointing strategy to active replication can allow us to identify the root-cause of such performance problems ahead of the underlying protocol's mechanisms; this advance detection can facilitate proactive recovery and lead to real-time fault-tolerant behavior.

Example 1: Propagation of performance problems. Due to the inherent coupling between nodes in a group communication protocol, faults that originate at a single node can propagate to other nodes in the system. For instance, the circulating token described in Section 2.1 can lead to performance side-effects. A slow node can retard the circulation of the token. The token's slowdown might lead a node to suspect a token-loss if it does not receive the token over a period of time (called the token-loss timeout, $T_{token-loss}$). This suspecting node will simply assume that the token loss was due to the crash of its preceding node in the ring, and will then initiate a membership change to eject the supposedly crashed node to form a new ring.

However, the slow node might continue sending messages. If one of these messages is seen by the new ring's members, the latter will initiate yet another membership-change to re-include the ejected node. Spread ensures that membership-changes converge, but a faulty node can trigger subsequent membership changes once the ring is formed. This sequence of repeated ring formations can repeat indefinitely, with the node-group membership continually thrashing and no useful application-level progress being made. Thus, a performance slowdown in a single node can degrade the entire system's responsiveness and availability.

Figure 1 shows the oscillating node-group membership when we injected a performance-degradation fault at node X by dropping 20% of its network traffic. Because $T_{token-loss}$ expires, a membership change is initiated to eject node X. The new ring contains nodes that respond within the membership-change timeout, T_{memb}. Node X somehow manages to respond within T_{memb} and is included in the new ring. Once in the new ring, node X "acts up" again, leading to yet another membership change. Only once did we see node X not respond within T_{memb},

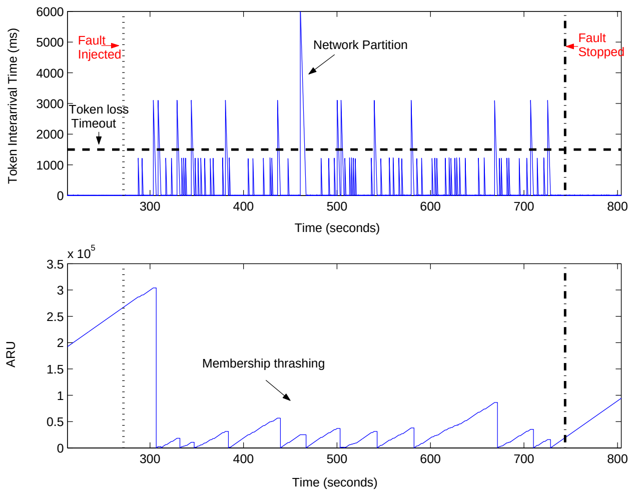


Figure 1. Thrashing membership due to a slow node (from the slow node's viewpoint).

thereby inducing a network partition, where only some nodes "saw" node X. Even in this case, node X triggered a new membership-change by sending a "foreign" message to the other partition. The membership oscillation is shown in the bottom half of Figure 1. The ARU (`all_received_upto`) counter on the y-axis indicates the sequence number upto which all reliable ordered messages have been received by all nodes in the ring, i.e., there are no gaps in the sequence numbers of messages received by nodes upto this point. The ARU resets to zero with each new ring, and increases monotonically while that ring exists (as seen by the saw-tooth graph in the bottom half of Figure 1).

Thus, a slow node is indistinguishable from a crashed one through the failure-detectors (i.e., timeouts) that group communication protocols use. A truly crashed node would not induce thrashing membership; it is the alive-but-limping node that is detrimental to the performance of the system. The strategy of tuning the failure-detectors, i.e., using larger timeouts, would makes us less vulnerable to thrashing membership, but would imply longer fault-detection latencies. Even with larger timeouts, a slow node can cause a performance degradation by triggering protocol-level flow-control mechanisms that reduce the non-faulty nodes' sending rates so as not to overwhelm a slow receiver.

Example 2: Proactive reconfiguration. BFT has mechanisms in place to detect the crash/slowdown of a leader but no explicit mechanisms in place to detect the crash/slowdown of a follower; the sytem will continue to function as long as the threshold (f) for the number of faulty nodes has not been violated. However, it might be possible to exploit protocol- and OS-level metrics to detect failures in the followers and proactively reconfigure the system to stay within the threshold, *f*. For instance, consider a replicated system that tolerates 2 arbitrary faults (i.e., with *3f+1 = 7* server replicas) where one of the follower nodes becomes faulty. Because the fault lies in a follower, no view change occurs but the system now reduces to tolerating only 1 arbitrary fault (because only 6 good replicas are left). By monitoring performance metrics, it might be possible to fingerpoint the follower and migrate it to another node, thereby sustaining the system's ability to tolerate 2 arbitrary faults.

3.1. Motivation

These examples show that, in the presence of performance problems, group communication protocols might not have sufficient knowledge to isolate the faulty node. In fact, a single faulty node can cause problems (e.g., performance slowdowns, missed deadlines) at even non-faulty nodes. Thus, any active replication strategy layered on top of these protocols might not mask failures adequately or in time. Fingerpointing techniques can complement the protocol's fault-tolerance mechanisms to provide a more responsive strategy.

Proactive reconfiguration might be possible where sufficient system information exists to enable us to fingerpoint the faulty node/process ahead of the protocol. For some kinds of failures, performance metrics can exhibit a discernible build-up (or pre-cursors) to the failure. The analysis of these metrics can lead to early failure detection (potentially before the protocol's timeouts expire, initiating membership changes) and, therefore, preemptive recovery and better availability. However, fingerpointing requires care in the presence of the protocols' coupling behavior: we need to be able to monitor the right system indicators or we might mis-diagnose the problem.

4. Fingerpointing Approach

Our system consists of an instrumentation framework, an anomaly-detection algorithm and a fingerpointing algorithm. The framework collects application-, OS- and protocol-level metrics at runtime and records them in a log that we subsequently analyze using our fingerpointing algorithm. In this paper, we perform our investigations on actively replicated systems implemented using Spread and BFT. For our evaluation of Spread, we use the MEAD replication middleware [13] with CORBA client-server applications. For BFT, we used the native active replication support packaged with the software. We used default membership timeouts (5 seconds) for both Spread and BFT.

We make the following assumptions in our experimental investigations: (i) our fault model covers process-crash, node-crash, message-loss and resource-exhaustion faults, (ii) we inject only a single, independent fault at a time into

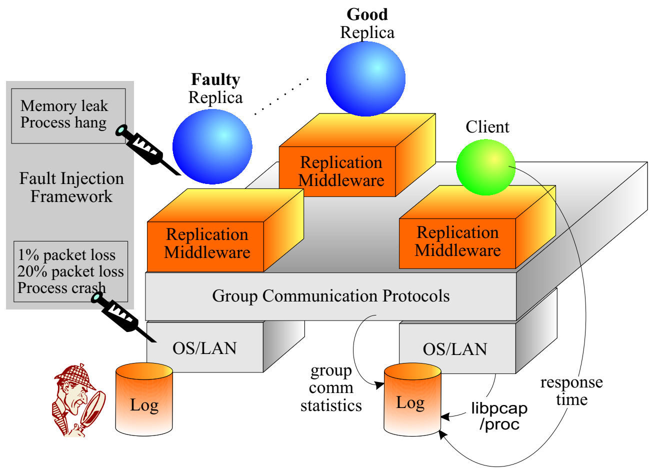


Figure 2. Instrumentation and logging.

one of the nodes, (iii) the client-server application has a constant workload and is deterministic, (iv) clocks are synchronized across nodes in order to correlate the node-level metric logs for the purpose of fingerpointing.

4.1. Instrumentation Framework

We collect the statistics of various performance metrics under both fault-free and injected faulty conditions. Our data is of two forms: time-series and event-series. With time-series data, the metric of interest, say, CPU usage, is sampled and recorded on a periodic basis. The event-series data arises because we gather information when specific events occur in the system, e.g., when the client receives a reply from the server. A node's total available memory, context-switch rate, CPU usage and network-traffic rate represent time-series data. The application- and protocol-level metrics represent event-series data. To enable the time-wise correlation of metrics across nodes, we convert all the event-series data (except for group-membership changes) into time-series data by aggregating these metrics over a specific time-period, e.g., by converting the client-side round-trip time into requests/sec. Our *unoptimized* instrumentation overhead is 28% for Spread and 12% for BFT, with the former being higher due to monitoring the token. Because our current fingerpointing granularity is the node, we focus on node-level metrics and do not consider process-level metrics.

OS-level metrics. We monitor node-level resource usage by retrieving the resource-usage statistics from each node's `/proc` pseudo-filesystem [9] every second. We monitor the following OS-level metrics on every node:

► CPU usage (%): Percentage of time that the node's CPU is busy executing both user and kernel tasks.

► Available memory (bytes): Sum of the node's free and the cached memory. Cached memory is used by Linux for the disk cache, and can be replaced quickly if a running/new program needs memory.

► Context-switch rate (per second): Number of context switches on the node in one second. A context switch occurs when a currently executing process is swapped out so that the CPU can execute a different process.

We monitor the network using the `libpcap` packet-capture facility, which provides a high-level interface to capture a node's incoming and outgoing network traffic. We record a timestamp for every new packet seen on the wire.

Application metrics: We monitor the response time in the client-side of the application. This is an event-series of the time intervals between the client's transmission of a request to, and its receipt of a reply from, the replicated servers. With Spread application, the client selects the first reply that it receives from any replica, while in BFT, the client waits for *f+1* identical replies, one from each replica.

Protocol metrics: For Spread, we monitor the token-interarrival time, the message-retransmission rate and membership changes. For BFT, we monitor the checkpoint frequency, the message-retransmission rate and membership changes.

4.2. Fault Injection

We injected a variety of performance-degrading faults at one of the nodes hosting a server replica in our replicated system through an interceptor that exploits the library interpositioning facilities [12] of the OS linker-loader.

► Memory leak: We inject a memory leak by bypassing the `free()` system call using the interceptor. To accelerate the rate of the leak, we modify our application to allocate/deallocate 96kB with each request, as a part of normal application behavior. This fault studies the effect of gradually loading a node and starving the protocols of memory.

► Process hang: We intercept the application's `read()` system call and block the application for several minutes. The fault investigates the effect of a slow receiver on the protocol's flow control.

► Abrupt crash fault: We abruptly kill one of the replicas.

► Packet-loss fault: We intercept the protocol's `send()` and `recv()` calls and randomly drop incoming and outgoing packets at packet-loss rates of 1% and 20% of the packets at the node. This fault investigates the effect of message retransmissions and network partitions in the protocol.

4.3. Anomaly Detection

We use a simple standard-deviation-based approach to detect anomalies in the metrics that we log. First, we designate an initial, fault-free training period, *trainingWin*, when we compute an initial mean, μ, and standard deviation, σ, for each metric. In our experiments, our *trainingWin* used 10% of the data samples. After the initial training, we adapt to changes in the metric's value by updating μ (and σ) based on the current observation plus a weighted previous μ (and σ), with λ being the weighting factor. We flag as anomalies any values that fall beyond upper ($+6\sigma$) and lower (-6σ) anomaly-detection thresholds (see Algorithm 1).

Algorithm 1 Anomaly Detection

```
trainingWin = T {length of fault-free training period}
$\mu$ = mean of metric over trainingWin
$\sigma$ = standard deviation of metric over trainingWin
$\lambda$ = 0.95 {weighting factor}
thresholdAnomaly = $6\sigma$ {anomaly-detection threshold}
anomalyWin = 7 {anomaly-detection window}
thresholdNoise = 3 {filters out noise from "true" anomalies}

for each metricSample after trainingWin do
  if $|\mu$ - metricSample$|$ < thresholdAnomaly then
    $\mu = \lambda\mu + (1-\lambda)metricSample$
    $\sigma = \sqrt{\lambda\sigma^2 + (1-\lambda)(metricSample - \mu)^2}$
  else
    flag anomaly in metric at current metricSample
  end if
end for
for each metricSample after trainingWin do
  if anomaly has been detected at the current metricSample then
    numAnomalies = number of anomalies in anomalyWin
    centered at current metricSample
    if numAnomalies < thresholdNoise then
      unflag anomaly in metric at current metricSample
    end if
  end if
end for
```

Next, we reduce the noise in the metrics by defining an anomaly-detection window, *anomalyWin*, and ignoring anomalies that fall below a certain anomaly-count, *thresholdNoise*, in that window. In our experiments, we used *anomalyWin* = 7 data samples, and *thresholdNoise* = 3 anomalies. These parameters were chosen because they yielded a false-positive rate of less than 5% in anomaly detection. We derive anomaly logs for the following metrics: memory, packets/sec, context-switches/sec and requests/sec. CPU usage both at the OS- and process-level displayed a high variance and proved to be unreliable for anomaly detection. In contrast, some metrics, such as memory usage, were fairly constant and our σ-based anomaly detection yielded overly tight thresholds with a high false-positive rate. Thus, for memory usage, we used an alternative, mean-based, approach, specifically, using a *thresholdAnomaly* of 0.5% of μ, instead of 6σ, to detect anomalies.

4.4. Fingerpointing

The anomaly-detection process in Section 4.3 serves as a preparatory phase for our fingerpointing algorithm. Due to the inherent coupling in group communication protocols, we fingerpoint the faulty node by comparing deviations in

Algorithm 2 Fingerpointing

```
fingerpointWin = 15 {fingerpointing window}
thresholdFingerpoint = 8 {number of anomalies in fingerpointWin needed to flag a problem}

for each fingerpointWin do
  for each server node do
    for each metric do
      num = number of anomalies in metric in fingerpointWin
      if $num > thresholdFingerpoint$ then
        flag problem in metric in node in fingerpointWin
      else
        ignore anomalies in metric in node in fingerpointWin
      end if
    end for
  end for
  if problem in any metric(s) is flagged in only one node then
    fingerpoint that node
  else if problem in metric(s) is flagged in multiple nodes then
    if there exists a node with anomalies in most number of metrics then
      fingerpoint that node
    else if there exists a node with most number of anomalous metrics and that has previously shown anomalies then
      fingerpoint that node
    else
      cannot fingerpoint the faulty node
    end if
  end if
  advance fingerpointWin
end for
```

behavior across the server nodes in the system, instead of focusing on the behavior of only a single node[1].

Our empirical evaluation suggests that the root-cause of the failure is either the node exhibiting the most anomalies in a given window of time or the node with an "odd-man-out" behavior, e.g., if a node displays a surge in context-switch rate while the other nodes display a dip in the same metric. Our fingerpointing algorithm is based on the assumption that the node with the most anomalous behavior is the root-cause of the failure. We outline our fingerpointing algorithm in Algorithm 2. To perform fingerpointing across the server nodes, we synchronize our anomaly logs across the nodes using timestamps. We then bin the anomaly data for each metric into fingerpointing-windows, *fingerpointWin*, of 15 data samples each, and count the number of anomalies in each bin. If more than *thresholdFingerpoint* entries in a bin are anomalous, we flag a potential problem with this metric for that node.

1 We ignore anomalies on the client's node because our faults are injected only on nodes hosting server replicas.

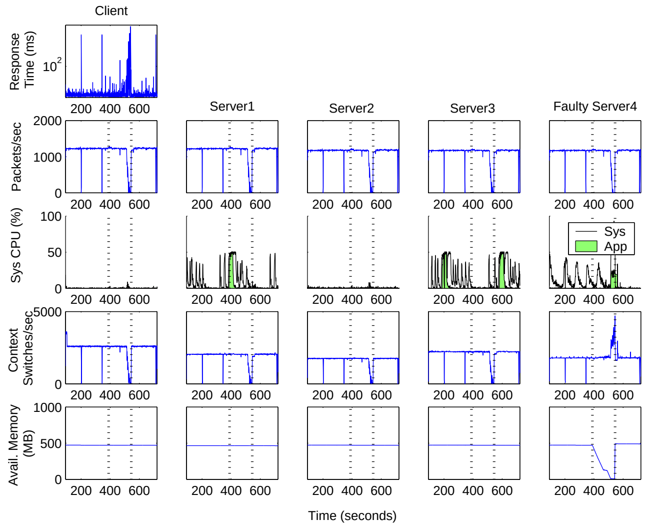


Figure 3. Metrics across nodes for a memory-leak fault at a Spread-hosted server.

5. Empirical Observations

Testbed. We conducted our experiments in the Emulab distributed testbed [15]. We used 5 nodes (850MHz processor, 256kB cache, 512MB RAM, RedHat Linux kernel 2.4.18) interconnected by a 100Mbps LAN. Our test application consisted of a simple actively-replicated client-server application, with one client and four server replicas, each on its own node. The client sends 1024 bytes of data to the server at a rate of ~50 requests/sec. Each experimental run covers 30,000 round-trip client requests and runs for ~10 minutes. We collect traces for the metrics identified in Section 4.1, and inject the 5 faults identified in Section 4.2. We run each experiment 5 times, yielding a total of 60 runs per experiment (i.e., 2 protocols * (5 faulty + 1 fault-free) runs * 5 times).

5.1. Insights from Data Traces

Independent failures, correlated manifestations. Figure 3 and Figure 4 show the progression of a memory-leak fault, injected at ~400 seconds, in the Spread-hosted `server4` and the BFT leader `server4`, respectively. In both cases, available memory is first metric to exhibit an anomaly. The memory leak eventually slows down `server4` due to increased paging and CPU activity as `server4` runs out of memory; `server4` finally crashes at ~600 seconds, and is subsequently restarted. Even though we have only one faulty replica, the client-side response time increases over the fault-injection duration, indicating that the client perceives *all* of the server replicas to have problems. Additionally, at the point of the crash, a large spike in client-side

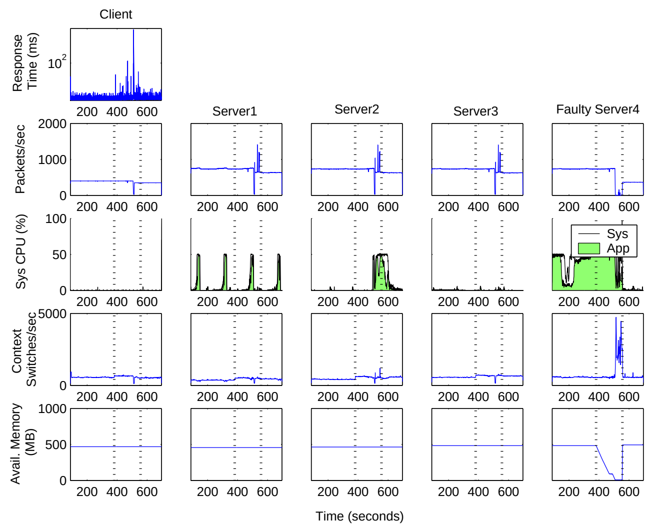


Figure 4. Metrics across nodes for a memory-leak fault at the BFT leader.

response time is observed, corresponding to a membership change in Spread or a view change in BFT.

In Spread, the memory-leak in `server4` results in a slowdown in the token's circulation, and a resulting drop in network-traffic and context-switch rates at even the non-faulty nodes. The client observes increased response times, even though this is an actively replicated server with the client simply picking the first response that it receives from any of the server replicas. We can clearly see the increasingly anomalous behavior in the metrics as the fault progresses. Note that the context-switch rate in the non-faulty nodes reduces while that at the faulty one increases. This suggests that we could refine our fingerpointing algorithm to examine whether anomalies are due an increase or decrease in metric values, rather than simply counting the sum of anomalies in a fingerpointing-window.

In the BFT case, performance degradations at the leader node can degrade the performance of the entire system. Performance slowdowns in the follower do not seem to disrupt the system as much but can be detected by our fingerpointing algorithm due to increased message retransmissions at the non-faulty servers. The new leader after the view change is `server3`. The pronounced drop in network-traffic at `server4` after the view change is due to increased paging at `server4`. However, `server4` continues requesting retransmissions, and hence, the increased network traffic at the non-faulty nodes.

5.2. Insights from Anomaly Detection and Fingerpointing

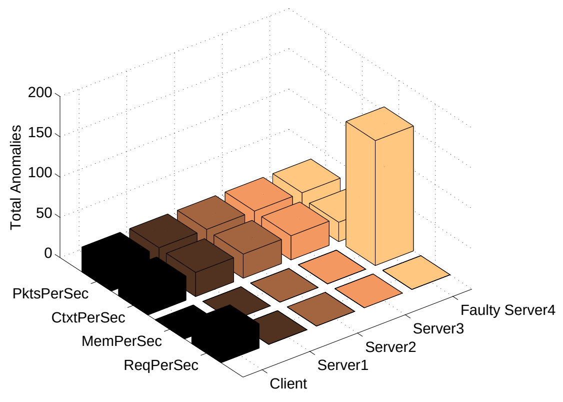
System-level metrics can assist in anomaly detection and fingerpointing. The context-switch rate on a node and the client's request rate seem to be closely correlated to the network-traffic (packets/sec) observed at the node. Faults (such as packet losses) that manifest solely on these metrics are difficult to fingerpoint because they lead to correlated fault-manifestations across all nodes. However, faults, such as memory leaks, that manifest on other metrics in addition to network traffic are easier to fingerpoint because they lead to noticeable differences in behavior across the faulty and non-faulty nodes (see Figure 4).

Figure 5(a) gives insight into our strategy that fingerpoints the node with the most anomalous metrics as the root-cause of the failure. For the memory leak in Spread, memory usage on `server4` exhibits the most anomalies from the time of fault-injection to node-crash, thereby leading to our fingerpointing `server4`. Note that we also flag anomalies in packets/sec, context-switch rate and requests/sec as the memory-leak progresses.

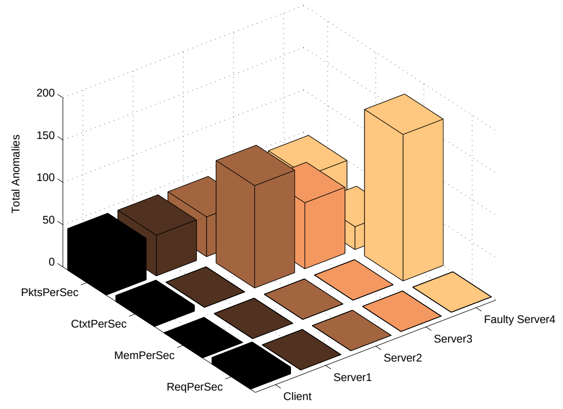
With the memory leak in the BFT leader in Figure 5(b), we similarly fingerpoint `server4` as the root cause. However, unlike Spread, the anomalies in the context-switch rate on the non-faulty `server2` and `server3` are high. We hypothesize that these are due to increased message retransmissions at the non-faulty servers and the view change before `server4` crashes. `server1` is not flagged for any anomalies although its context-switching behavior is similar to `server2` and `server3` (see Figure 4) because its context-switching activity does not deviate enough to exceed our anomaly-detection thresholds.

Although `server4`'s context-switch rate was significantly higher than that of the non-faulty nodes, its anomaly count was lower than that of the non-faulty nodes because we use a single threshold to detect anomalies. This suggests that the accuracy and precision of our fingerpointing might be improved if we weighted anomalies by the extent to which they violate our thresholds. For example, we could assign a higher weight in the fingerpointing algorithm to a metric that violates its threshold by 9σ as opposed to 6σ. Figure 5(c) represents the anomaly count for a memory leak in the BFT leader. Again, `server4` is fingerpointed.

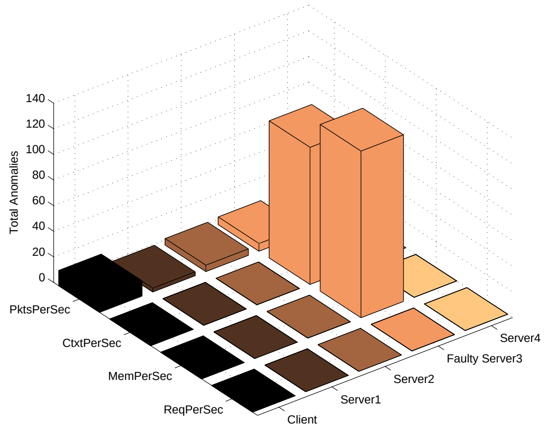
Figure 6 shows the progress of our fingerpointing algorithm for the memory leak in the Spread case. We observe a false positive (since an anomaly is flagged even before the fault is injected) in requests/sec, packets/sec and context-switch rate at ~150 seconds. This anomaly is flagged on all nodes in the system due to the coupling of the token-ring protocol. Anomalies in memory are flagged once we inject the fault. At ~30 seconds before `server4`'s crash, just when `server4`'s node starts to run out of memory, we start to flag anomalies in packets/sec, context-switch rate and requests/sec because of the increased paging activity and the slowed-down token. The anomalies in memory on `server4` persist for some period even after the node recovers due to the altered memory usage at `server4`'s node



(a) Spread



(b) BFT Leader



(c) BFT Follower

Figure 5. Sum of anomalies from our fingerpointing algorithm in the window of time between fault injection to node-crash, for a memory leak in Spread and BFT.

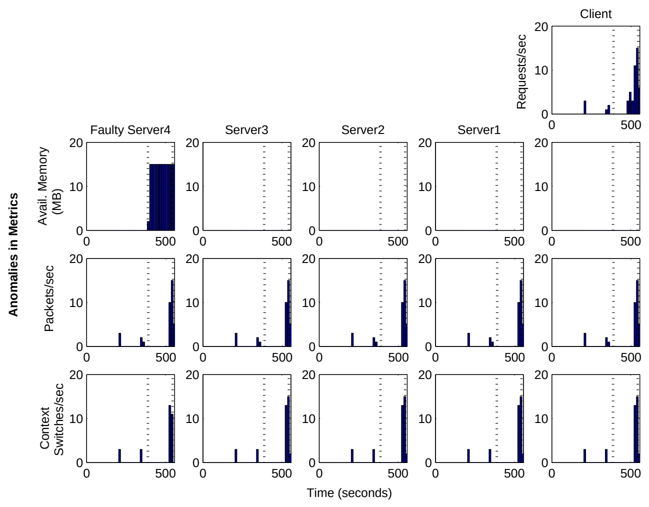


Figure 6. Progress of fingerpointing for the case shown in Figure 3.

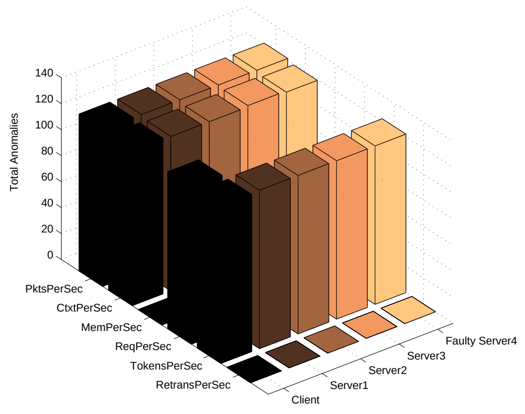
after the membership change. Our adaptive algorithm does not fold in the altered memory profile and instead flags it as a series of anomalies; this suggests a need to retrain our algorithm after membership changes.

5.3. Insights from Protocols

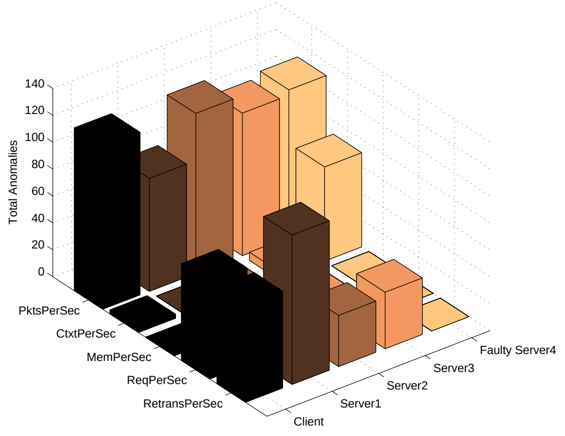
The underlying protocol (token ring vs. quorum) and its implementation (library vs. daemon) influence the result of our fingerpointing. To illustrate this, we run further experiments to examine protocol-level metrics for packet-loss faults since these resulted in the highest number of false positives in our fingerpointing algorithm (see Table 1).

Figure 7(a) shows the anomaly count over the duration of a 20% packet-loss fault in `server4` in the Spread case. The circulating token leads to correlated manifestations of the fault on packets/sec, context switch rate and requests/sec. Packets/sec drops due to the slower token circulation, leading to a corresponding drop in context-switch rate across all nodes because every packet processed by Spread requires a context switch due to Spread's daemon architecture. While we can detect that there is a problem in the system because of the correlated anomalies on OS-level metrics, we cannot fingerpoint the guilty node.

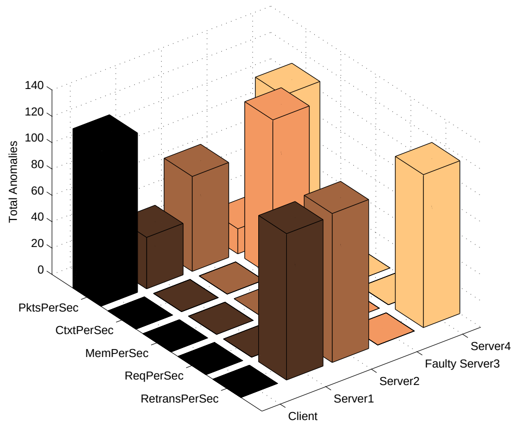
Despite the high packet-loss rate, the number of anomalies in message retransmissions was zero! Examining the absolute number of retransmissions during this 2-minute fault-injection duration, we observed 13 retransmissions at the client and 4 retransmission requests at the faulty server. These retransmissions seemed to be insufficient to trigger our anomaly-detection algorithm. We hypothesize that this is because (i) a low retransmission rate results because our system has moderate load and we are more likely to drop a token than a message, and (ii) Spread sends a token twice



(a) Spread



(b) BFT Leader



(c) BFT Follower

Figure 7. Sum of anomalies from our fingerpointing algorithm in the window of time between fault injection to node crash for a 20% packet-loss fault in Spread and BFT.

when it suspects a lossy network. This suggests that retransmission rate might be a useful fingerpointing metric at higher loads or higher packet-loss rates. During the same duration, there were 2 membership changes and the faulty server kept getting included in the formation of the new ring because it responded within the membership-change timeouts. This thrashing-membership behavior hinders the fingerpointing process.

Figure 7(b) and Figure 7(c) show the anomaly count for the 20% packet-loss fault at the BFT leader and follower. Our fingerpointing algorithm was able to fingerpoint the faulty BFT leader using OS-level metrics alone. However, the algorithm was unable to diagnose the root-cause in the BFT-follower case because the follower is not in the critical path of client-server communication. If we examine the protocol-level metrics in addition to the OS-level metrics, we are able to fingerpoint the guilty BFT follower using an odd-man-out approach (rather than the anomaly-count approach).

5.4. Results of Fingerpointing Algorithm

While we discussed memory-leak and packet-loss faults in detail, Table 1 summarizes our results for fingerpointing various faults through OS-level metrics. In this section alone, we use the term "window" to refer to the fingerpointing-window. The following criteria help to evaluate the effectiveness of our approach:

▶ False positive rate: Average fraction of windows where we incorrectly fingerpoint a node, prior to fault injection. Because our false-positive rate was zero, we exclude this information from the table.

▶ True positive rate (Accuracy): Average fraction of windows, over the fault-injection duration, where we correctly fingerpoint the guilty node. We count all fingerpointing windows from the time of fault-injection up to 3 windows after the fault stopped.

▶ False negative rate: Average fraction of windows, over the fault-injection duration, where fingerpointing was unsuccessful, i.e., we either did not fingerpoint any nodes or we fingerpointed innocent nodes.

▶ Obscuring rate: Average fraction of windows, over the fault-injection duration, where we fingerpointed both the guilty node and innocent ones, i.e., we could not tell which node was the true root-cause.

▶ Fault detection latency (secs): Average time from fault-injection until the problem was detected (but not fingerpointed) on any node in the system.

▶ Fingerpointing latency (secs): Average time from fault-injection until successful fingerpointing of the guilty node. A '–' indicates that we did not fingerpoint the faulty node.

▶ Window of opportunity (secs): Average time elapsed from the time we fingerpoint the faulty node to the time the node crashes or fault-injection stops. A positive value is in-

Fault	Protocol	True Positive Rate	False Negative Rate	Obscuring Rate	Detection Latency (sec)	Fingerpoint Latency (sec)	Window of Opportunity (sec)
Memory Leak	Spread	0.95	0.05	0	14.30	14.30	+120.70
	BFT Leader	0.70	0.03	0.27	13.78	58.78	+61.22
	BFT Follower	1.0	0	0	13.95	13.95	+76.05
Hang	Spread	0.76	0.24	0	40.75	40.75	+49.25
	BFT Leader	0.95	0.05	0	13.53	13.53	+121.47
	BFT Follower	0	0	1	13.68	–	–
Crash	Spread	0	1	0	12.09	–	–
	BFT Leader	0.89	0.11	0	13.60	13.60	-13.60
	BFT Follower	0.67	0	0.33	13.42	28.2	-28.2
1 % Pkt Loss	Spread	0	0.31	0.69	15.30	–	–
	BFT Leader	0	0.74	0.27	14.13	–	–
	BFT Follower	0	0.23	0.74	9.14	–	–
20% Pkt Loss	Spread	0	0.28	0.72	16.35	–	–
	BFT Leader	0	0.23	0.78	9.14	–	–
	BFT Follower	0.89	0.11	0	14.08	14.08	+75.92

Table 1. Results of fingerpointing approach with OS-level metrics.

dicative of how much time we have for proactive recovery. A negative value indicates that we fingerpointed the faulty node after the node crashed. A '–' indicates that we did not fingerpoint the faulty node.

Fingerpointing out-performs protocols: Our fingerpointing algorithm could diagnose certain performance failures ahead of the protocol, and provide a window of opportunity for proactive recovery that could mitigate the impact of faults on the system's real-time behavior. We correctly diagnosed memory-leaks for both protocols, process-hangs in Spread, and crash and 20% packet-loss faults in the BFT follower. The protocol's mechanisms were able to diagnose the memory-leak for Spread and the BFT leader; the process-hang in Spread and faults in the BFT follower were not diagnosed by the protocol's mechanisms.

The process-hang fault highlights a difference between the daemon (Spread) and. library (BFT) protocols. In Spread, a process hang does not affect the progress of the underlying daemon that represents the protocol. Our algorithm can detect and fingerpoint the fault because the process-hang fault induces a message backlog in the faulty process. We fingerpoint the fault ahead of the protocol's mechanisms because the backlog causes the faulty node's memory profile to change. Once the faulty process' backlog reaches a certain limit, Spread disconnects the process to avoid being hindered by a slow receiver; this local disconnection has little effect on the other nodes. A process hang in the BFT leader provokes a view-change once the client detects the non-responsiveness of the leader. Thus, the protocol's mechanisms diagnose the fault ahead of our fingerpointing algorithm.

Protocols out-perform fingerpointing: The protocol was better at detecting faults that abruptly stop its progress, e.g., crash fault in Spread and BFT leader, and process-hang in BFT leader. The process-crash fault also illustrates the difference between daemon and library protocol implementations. In Spread, this fault is neither detected nor diagnosed by our algorithms in 2 out of 5 runs. We incorrectly fingerpoint nodes in the remaining 3 runs. This occurs because a process-crash is a local disconnection from the Spread daemon and does not perturb the system much, i.e., Spread diagnosed the fault 0.05 seconds after crash. On the other hand, crash of the BFT leader results in a view change 5.3 seconds after the crash.

Both protocols and fingerpointing are inadequate: The process hang in the BFT follower was detected, but mis-diagnosed, by our fingerpointing algorithm. This occurred due to a drop in network traffic across all nodes because the hung process stops sending messages. This drop in network traffic is most pronounced at the faulty node, but our current algorithm is unable to detect this because we do not consider the extent to which a metric deviates from the threshold. The protocol's mechanisms did not diagnose the process-hang in the follower.

The packet-loss fault in both Spread and BFT was detected but was difficult to fingerpoint using the OS-level metrics that we monitored. This fault is especially difficult to diagnose in Spread because the circulating token causes correlated fault-manifestations on all nodes. The factors underlying the fingerpointing of this fault were discussed in greater detail in Section 5.3 for both Spread and BFT. The packet-loss faults did not trigger membership changes with the exception of the 20% packet-loss in Spread (1-2 membership changes occurred during each run). Even in these cases, the membership mechanisms could not diagnose the fault either because the faulty node responded within the membership timeouts and was included in the new ring. We hypothesize that some of these faults might be diagnosable if we increased the level of protocol instrumentation.

6. Related Work

Current research in application-level root-cause analysis centers on identifying the faulty components along the causal request path. Aguilera *et al.* [1] isolate performance problems by treating the system nodes as “black-boxes” and using message-level traces to passively ascribe performance problems to specific nodes on the causal request path. Sailer *et al.* use request-dependency tracking to identify the root-cause of response time violations in e-commerce applications. Magpie [4] captures the control-path and resource demands of application requests as they are serviced across components and nodes in a distributed system. They use behavioral clustering to construct representative workload models that can be used for anomaly detection. Cohen et al. [8] present a (not as tightly coupled) system that uses machine learning to identify system metrics that are most correlated with SLO violations. Based on this, they extract indexable failure signatures for root-cause analysis.

However, components along the causal request path whose behavior is identified as anomalous may not always be the source of the problem. This may occur due to hidden dependencies between nodes that are not directly related to the request call-graph. Our approach provides insight on how to diagnose such failures in distributed, replicated systems.

Kiciman and Fox [11] determine the cause of partial failures in J2EE environments by monitoring the flow of requests through the system. They build reference models of the component interactions based on historical behavior as well as the behavior of replicated peers. Our approach also uses peer comparison to diagnose faults in a system. However, we focus on diagnosing performance problems, as opposed to partial failures in the application.

Several fault-injection studies have been conducted on group communication systems. Joshi et al. [10] study the unavailability induced by group communication protocols during membership changes triggered by crash faults. Ramasamy et al. [14] investigate the overhead of providing intrusion-tolerance mechanisms in group communication protocols. Our approach, on the other hand, investigates the propagation of performance problems in replicated applications that rely on an underlying group membership protocol. We also present an approach for diagnosing these problems even in the absence of a membership change.

Basile et al. [5] conducted a fault-injection study of the Ensemble group communication system. They injected memory- and message-corruption faults and showed that error propagation can occur in group communication systems. We use a different fault model (namely, performance-degrading failures). We show that performance problems can propagate in a group communication system and then present a fingerpointing approach.

Alvarez et al [2] used simulation-based testing to validate two group membership protocol implementations. They injected both crash and performance failures to determine whether the systems violated their specification. Our fault-injection is not targeted at testing the correctness of the protocols, but rather at determining whether we can improve system availability and provide better fault containment through fingerpointing.

7. Conclusion

We investigated the influence of an underlying group communication protocol on the ability to detect and fingerpoint faulty nodes in a distributed, replicated system. We conducted our investigations on two different group communication protocols, namely, the token-protocol, Spread, and the quorum-based Castro-Liskov BFT. Our empirical evaluation shows that these protocols can aggravate root-cause analysis because their innate coupling across nodes causes fault-manifestations to “travel” across the distributed system. We present a protocol-agnostic anomaly detection and fingerpointing approach for identifying the root-cause of performance problems. The goal of our fingerpointing approach is to complement the group communication protocols failure detection mechanisms, and to provide better system availability and enable proactive recovery in the face of performance failures.

Our main insight is that, in order to diagnose performance problems in replicated systems using a consensus-based protocol, we need to compare anomalous behavior across nodes because focusing only on anomalies on a single node can lead to misdiagnosis due to the inherent coupling in these systems. Furthermore, because group communication protocols involve network-intensive (message-passing) coordination, faults (e.g., packet losses) that manifest solely on network-related metrics are difficult to diagnose using a black-box approach alone because these faults lead to correlated manifestations across the entire system.

The daemon vs. library implementation of the protocol also influences failure-masking and fingerpointing. Spread is a daemon-based implementation therefore faults that manifest solely at the application-level are better contained and lead to light-weight process-group membership changes. BFT is a library-based implementation therefore the application is integrated with the group membership service. Therefore, an application-level fault can cause heavyweight processor-group membership changes. Daemon-based architectures also provide the possibility of finer grained fingerpointing in which we could use process-level metrics to distinguish between faults at the application and faults in the group membership service.

Group communication protocols (e.g., BFT) that assign different roles to processes, such as leader and follower roles, can ease the task of fingerpointing because the asymmetry can help to identify the root-cause of the failure. Faults in the leader are easier to fingerpoint with a black-box approach, while those in the follower require additional help from protocol-level metrics. For example, if we notice a performance problem that propagates to all the nodes in the system, we can assume that the most likely culprit is the current leader of the group. Faults in follower nodes are more difficult to diagnose and may require further instrumentation of GCP metrics. In token-ring protocols, faults that hinder the progress of the protocol, e.g., by retarding the token, will result in the correlated manifestation of the fault on all of the nodes in the system, making fingerpointing difficult.

As a part of our future work, we intend to explore finer-grained fingerpointing using process-level metrics. We would also like to investigate the effect on fingerpointing of varying the protocol's configuration, e.g., the membership timeouts. We are currently developing a dynamic fingerpointing algorithm that deals with varying workloads (e.g., changing number of clients and request rates), and that can distinguish between a true anomaly and a change in workload. We would like to evaluate the effect of gossip-based protocols on real-time fault-tolerant operation, as well as the overheads and impact of running our fingerpointing algorithm online.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Symposium on Operating Systems Principles*, pages 74–89, Boston Landing, NY, October 2003.

[2] G. A. Alvarez and F. Cristian. Simulation-based test of fault-tolerant group membership services. In *Annual Conference on Computer Assurance*, pages 129–138, Gaithersburg, MD, June 1997.

[3] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *International Conference on Dependable Systems and Networks*, pages 327–336, New York, NY, June 2000.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 259–272, San Francisco, CA, December 2004.

[5] C. Basile, L. Wang, Z. Kalbarczyk, and R. Iyer. Group communication protocols under errors. In *Symposium on Reliable Distributed Systems*, pages 35–44, Florence, Italy, October 2003.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, USA, February 1999.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[8] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Symposium on Operating Systems Principles*, pages 105–118, New York, NY, USA, October 2005.

[9] R. Faulkner and R. Gomes. The process file system and process model in UNIX System V. In *USENIX Winter Technical Conference*, pages 243–252, Dallas, TX, January 1991.

[10] K. R. Joshi, M. Cukier, and W. H. Sanders. Experimental evaluation of the unavailability induced by a group membership protocol. In *European Dependable Computing Conference*, pages 140–158, Toulouse, France, October 2002.

[11] E. Kiciman and A. Fox. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks: Special Issue on Adaptive Learning Systems in Communication Networks*, 16(5):1027– 1041, September 2005.

[12] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, San Francisco, CA, 2000.

[13] P. Narasimhan, T. A. Dumitraş, S. M. Pertet, C. F. Reverte, J. G. Slember, and D. Srivastava. MEAD: Support for real-time fault-tolerant CORBA. *Concurrency and Computation: Practice and Experience*, 17(12):1527–1545, October 2005.

[14] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, and W. H. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. In *International Conference on Dependable Systems and Networks*, pages 229–238, Bethesda, MD, 2002.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002.