Understanding Self-healing in Service-Discovery Systems

C. Dabrowski
U.S. National Institute of Standards and Technology
Gaithersburg, MD 20899
+1 (301) 975-3249
cdabrowski@nist.gov

K. Mills
U.S. National Institute of Standards and Technology
Gaithersburg, MD 20899
+1 (301) 975-3618
kmills@nist.gov

ABSTRACT

Service-discovery systems aim to provide consistent views of distributed components under varying network conditions. To achieve this aim, designers rely upon a variety of self-healing strategies, including: architecture and topology, failure-detection and recovery techniques, and consistency maintenance mechanisms. In previous work, we showed that various combinations of self-healing strategies lead to significant differences in the ability of service-discovery systems to maintain consistency during increasing network failure. Here, we ask whether the contribution of individual self-healing strategies can be quantified. We give results that quantify the effectiveness of selected combinations of architecture-topology and recovery techniques. Our results suggest that it should prove feasible to quantify the ability of individual self-healing strategies to overcome various failures. A full understanding of the interactions among self-healing strategies would provide designers of distributed systems with the knowledge necessary to build the most effective self-healing systems with minimum overhead**.**

Categories and Subject Descriptors

D.1.3 **[Concurrent Programming]**: Distributed programming

General Terms

Algorithms, Measurement, Performance, Design, Reliability, Experimentation.

Keywords

Architecture, Self-Healing Systems, Self-Repairing Systems, Service Discovery.

1. INTRODUCTION

Growing deployment of wireless communications, implying greater user mobility, coupled with proliferation of personal digital assistants and other information appliances, foretell a future where software components can never be quite sure about the network connectivity available, about the other software services and components nearby, or about the state of the network neighborhood a few minutes in the future. In extreme situations, as found for example in military applications [1], software components composing a distributed system may find that cooperating components disappear due to physical or cyber attacks or due to jamming of communication channels or movement of nodes beyond communications range. In such volatile environments, service discovery protocols enable distributed components to rediscover lost components or to find other components that provide essential services needed to accomplish critical tasks. To do this, service discovery systems include self-healing strategies to mitigate, detect, and recover from failures.

Service discovery protocols rely on several self-healing strategies. Architecture, which defines the logical components and relationships that compose a system, coupled with topology, which specifies the number and placement of components in a system, can be used in combination to mitigate the effects of failures by increasing system redundancy. Failure detection techniques, which typically include monitoring of periodic announcements and bounded retries (and resulting exceptions), allow components to estimate uncertainty regarding the state of cooperating components or regarding the intervening network path. Recovery techniques, which include application-level persistence and soft state, define actions a component can take to address suspected failures. Consistency-maintenance mechanisms, which include notification and polling, provide a means to maintain synchronized state among distributed components by propagating state changes to remote components.

In previous work, we used architectural models to investigate the behavior of various service-discovery systems under increasing communication failure [6] and message loss [7]. Our investigations yielded quantitative measures for the effectiveness, responsiveness, and efficiency of alternate system designs. We considered various combinations of architecture, topology, and consistency-maintenance mechanisms, however we did not vary failure recovery techniques.

In this paper, we extend our approach to quantify the contribution of failure recovery techniques in order to provide a more complete picture of the actions of individual self-healing strategies within

service discovery systems. We focus our investigation on four combinations of failure-detection and recovery technique while limiting other variables to include only two architecture-topology combinations and one consistency-maintenance mechanism. We examine system behavior under increasing communication failure. We use the same Rapide [4] models of service-discovery systems that we used in our previous research. Our models are based on two specifications: Jini™ Networking Technology [2] and Universal Plug-and-Play [3]. We adapted self-healing strategies from these specifications.

The remainder of the paper is organized as four sections. In the first section, we provide an overview of the self-healing strategies used in service-discovery systems. The second section gives a quantitative summary of the overall effectiveness of various combinations of self-healing strategy, when used to maintain consistent state among distributed components as the duration of communication failures increases. In the third section, we investigate and quantify the contribution of failure detection and recovery techniques to overall system effectiveness. In the conclusions, we discuss the feasibility and desirability of gaining a full understanding of the interactions among self-healing strategies for adaptive distributed systems.

2. DISCOVERY SYSTEMS AND SELF-HEALING

Service discovery systems enable distributed software components to discover each other, and to determine if discovered components meet specific requirements. Discovery protocols include *consistency-maintenance mechanisms*, which can be used to disseminate changes in component availability and status, and to maintain, within some time bounds, a consistent view of components in a network. *Failure-detection and recovery techniques* enable components to detect and react to network changes by restoring communications with remote components or by locating alternate components. A number of different designs have been proposed for service-discovery systems. For example, a team at Sun Microsystems designed Jini Networking Technology, a general service-discovery mechanism atop Java™. As another example, a group from Microsoft and Intel conceived Universal Plug-and-Play (UPnP) to provide plug-and-play components for distributed systems.

2.1 Architecture and Topology

Our analysis of six distinct discovery systems revealed that most designs use one of two underlying architectures: two-party and three-party. A two-party architecture consists of two components types: service manager (SM) and service user (SU). Figure 1 shows a two-party architecture deployed in a six-component topology: one SM and five SUs. A three-party architecture adds a third component type: service cache manager (SCM). The three-party architecture allows for multiple SCMs to mitigate the effect of failures (passive self-healing). Figure 2 shows a three-party architecture with one SM, five SUs, and up to two SCMs. A SM maintains a database of service descriptions (SDs), where each SD encodes the essential characteristics of a particular service. A SU seeks SDs maintained by SMs that satisfy specific requirements. Where employed, the SCM operates as an intermediary, matching advertised SDs of SMs to SD requirements provided by SUs.

To animate our two-party model, we incorporated behaviors from the UPnP specification. Upon startup, each SU and SM seeks to discover other, relevant components within the network neighborhood. In a lazy-discovery process, each SM periodically announces the existence of its SDs over the UPnP multicast group. Upon receiving these announcements, SUs with matching requirements request copies of the desired SDs from the SM. The SU stores SD copies in a local cache. Alternatively, the SU may engage in an aggressive-discovery process by transmitting SD requirements, as Msearch queries, on the UPnP multicast group.

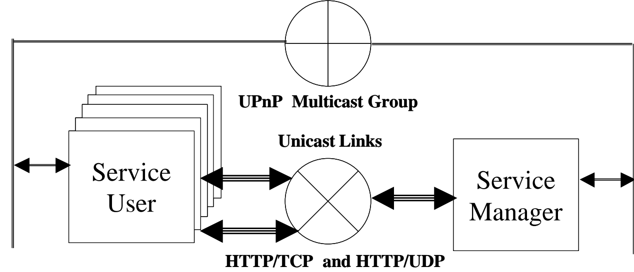

Figure 1. Two-party service-discovery architecture with five service users (SUs) and one service manager (SM).

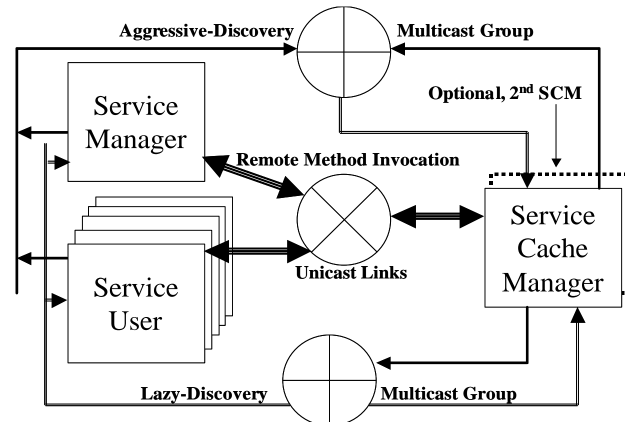

Figure 2. Three-party service-discovery architecture with five service users (SUs), a service manager (SM), a service cache manager (SCM), with an optional 2nd SCM.

Any SM holding a SD with matching requirements may respond directly to the SU. The SU may then request a copy of the relevant SDs, caching them locally. To maintain a SD in its local cache, a SU expects to receive periodic announcements from the relevant SM at a specified interval, known as a Time-to-Live, or TTL (or it must receive replies to its Msearchs within the TTL). Otherwise, the SU may discard the SD.

To animate our three-party model, we chose behaviors described in the Jini specification. In Jini, the discovery process focuses upon discovery by SMs and SUs of any intermediary SCMs that exist in the network neighborhood. Upon initiation, a Jini component enters aggressive discovery, where it transmits probes on the aggressive-discovery multicast group at a fixed interval for a specified period or until it has discovered a sufficient number of SCMs. Upon cessation of aggressive discovery, a component

enters lazy discovery, where it listens for announcements sent at intervals by SCMs. Once discovery occurs, a SM deposits a copy of the SD for each of its services on the discovered SCM for a specified length of time, or TTL. To maintain a SD on the SCM beyond the TTL, a SM must refresh the SD; otherwise it is purged. The SCMs match SDs provided by SMs to SU requirements, and forward matches to SUs.

2.2 Consistency-Maintenance Mechanisms

After initial discovery and information propagation (through SDs), SUs can use consistency-maintenance mechanisms to obtain updates to SDs for discovered services. We consider two basic mechanisms: notification and polling. In polling, a SU periodically sends queries to obtain up-to-date information about a previously discovered SD. In a two-party architecture, the SU issues the query directly to the SM from which the SD was obtained, and receives a response. In a three-party architecture, polling consists of two processes: 1) a SM propagates an updated SD to each SCM where the SD was originally cached and 2) each SU periodically queries relevant SCMs.

In notification, immediately after an update occurs, a SM sends events that announce a SD has changed. To receive events about a SD, a SU must first register for this purpose. In the two-party architecture, the SU requests registration with a SM. The request, if accepted, is retained for a TTL, which may be refreshed with subsequent requests from the SU. In a three-party architecture, a SU registers with a SCM to receive updates. The SCM grants event registrations for a TTL, which may be refreshed. When a SD is updated, the SM first propagates the update to all SCMs on which it deposited the SD; each SCM then forwards the event to all SUs registered to receive updates to the SD.

2.3 Failure-Detection Techniques

In a hostile military or emergency response environment, faults may arise due to enemy jamming or other interference, congestion, physical severing of cables, improperly configured or sabotaged routing tables, or multi-path fading as nodes move across a terrain. In this paper, we consider communication failure. Node communication may fail fully (both transmit and receive) or partially (either transmit or receive). All outbound messages from an interface will be lost when the transmitter fails, while all inbound messages will be lost when the receiver fails.

To detect failures, discovery systems use a combination of two techniques: monitoring periodic announcements and bounded retries (and resulting exceptions). Discovery protocols specify periodic transmission of key messages. Listeners can monitor these messages; much in the same way a heartbeat is monitored to assess the health of a patient. For example, as described above, both Jini and UPnP provide for periodic announcements of the availability of essential resources. Failure to receive scheduled announcements may indicate that the announcing entity has failed or that the network path is blocked. In other situations, software components send messages using reliable communication protocols, which persistently resend unacknowledged messages up to some bound, issuing a remote exception (REX) if the bound is exceeded. Failure detection allows components to employ recovery techniques.

2.4 Recovery Techniques

Discovery systems generally support two recovery techniques: soft-state and application-level persistence. Periodic announcements convey *soft* information about essential state, which a receiver can cache for a period of time, consistent with the expected announcement or heart rate. Each new re-announcement, or heartbeat, may convey updated state information; thus, the receiver overwrites previously cached state with state arriving in the latest announcement, or heartbeat. When the heartbeat fails, the receiver discards the cached state. When the heartbeat resumes, the receiver recovers the latest state. For example, upon failure of heartbeat messages sent by Jini SMs to refresh SDs cached on SCMs, the SD is discarded. The same occurs upon failure of periodic refreshes of notification registrations in both Jini and UPnP. Similarly, UPnP SUs may commence periodic *Msearch* queries after failure by a SM to refresh a SD within the TTL, which causes the SU to discard knowledge of the SM. Once a SU regains its desired SD, the related *Msearch* queries cease. This method is also employed when, after an initial aggressive discovery phase, Jini SCMs enter lazy discovery where they announce themselves every 120s. This ensures rediscovery of the SCM by SMs and SUs within 120s after a fault is rectified.

When failure-detection leads to a REX, discovery systems generally expect application software to initiate recovery, guided by an application-level persistence policy. In our models, depending on the situation, we implement three different persistence policies: (1) ignore the REX, (2) retry the operation for some period, and (3) discard knowledge. A SU can ignore a REX received in response to an attempted poll, because the query recurs periodically. In our models, two-party SMs and three-party SCMs also ignore a REX received as a result of attempted notifications. This behavior, which is described in both the Jini and UPnP specifications, depends upon reliable lower-layer protocols to provide robustness for events. In other cases, the retry policy attempts to recover from transient failures by resending a message (for which it has received a REX) after a nominal delay. The discard policy, which occurs following repeated failure of a retry, relies upon monitoring periodic soft-state announcements to recover from more persistent failures. As indicated above, in the two-party model, the SU discards the SM and its related SDs after failure to receive announcements from the SM within the TTL. In Jini, the specification states that a discovering entity *may* discard a SCM with which it cannot communicate. In our three-party model, a SM or SU deletes a SCM if it receives only REXs after attempting to communicate with the SCM over a 540-s interval. After discarding knowledge of a SM (UPnP) or SCM (Jini), all operations involving the node cease until it is rediscovered by monitoring periodic announcements (through either lazy or aggressive discovery).

3. EFFECTIVENESS OF SELF-HEALING

In previous work, we investigated the effectiveness of selected self-healing strategies when attempting to maintain synchronized state among distributed components during communication failure [6] and message loss [7]. We compared combinations of two- and three-party architectures and topologies (as shown in Figures 1 and 2), together with different consistency-maintenance mechanisms (notification or polling). In each combination, we used the same failure-detection (monitoring periodic

announcements and bounded-retries) and recovery (soft state and application-level persistence) techniques (see Table 1). We measured effectiveness (as the probability that a node achieves state synchronization) for increasing failure rates. Here we summarize our findings for effectiveness in the face of communications failure. Figure 3 shows effectiveness for each combination as communication-failure rate increases to 75%.

Failure Detection and Recovery Technique		Two-Party Architecture (UPnP)	Three-Party Architecture (Jini)
Heartbeat and Soft State	Lazy Discovery	SM: announces with *n* (3+2*d*+*k*) messages every 1800 s	SCM: announces every 120 s
	Aggressive Discovery	SU: issues *Msearch* every 120 s (after purging SD)	SU and SM: issue seven probes (at 5 s intervals) only during startup
Bounded Retries and Application-Level Persistence Policy	Ignore REX	SU: Pull Query (After Initial Pull) SM: Push Event	SU: Pull Query SCM: Push Event
	Retry after REX	SU: Initial Pull Query retry in 180 s (retries ≤ 3) Push Event Registration retry in 120s	SM: depositing or refreshing SD copy on SCM retry in 120s SU: Push Event registration or refresh on SCM retry in 120 s
	Discard Knowledge	SU: purge SD after failure to receive SM announcement within 1800 s	SU and SM: purge SCM after 540 s of continuous REX

Table 1. Summary of self-healing strategies included in our models.

Our previous papers provided qualitative explanations (based on analysis of execution traces) regarding the contributions of each self-healing strategy to measured differences in effectiveness. Here, we summarize our main findings for communications failure. Figure 3 indicates a rough similarity in effectiveness for all combinations; however, within these ranges, there are also significant differences. We attribute similarity in effectiveness to the fact that we employ similar failure-detection and recovery techniques in all combinations. The graph contains several eccentricities, in the form of saw-tooth behaviors. For example, two-party notification suffers a significant drop in effectiveness between 5% and 25% failure rate. This occurs because notifications rely on underlying reliable communication protocols to achieve robustness. When these protocols fail (as would be likely in case of communication failure), notifications are lost. The application software then relies upon detection of failure of periodic announcements (heartbeat) and restoration through initiation of recovery actions. Unfortunately, in UPnP the lazy-discovery announcement occurs no more frequently than every 1800s. Between 5% and 25% failure rate, there exists a substantial likelihood that communication failure is corrected prior to the next announcement. In such cases, an aggressive-discovery announcement (120-s interval) is not initiated, and state contained in the notification remains lost. As the failure rate increases, coincidence of announcement failure and notification failure becomes more probable, leading to initiation of the aggressive-discovery announcements, which eventually recovers state contained in the lost notification. Jini does not suffer as much from this phenomenon for two reasons. First, in Jini the lazy-discovery announcements occur at a 120-s interval. Second, Jini SMs exhibit some persistence when attempting to propagate SDs to SCMs. In selected cases, this persistence causes the SCM to periodically retry notifications.

Despite the dominance of failure-detection and recovery techniques, our results show that certain combinations of architecture, topology, and consistency-maintenance mechanism contribute to differences in effectiveness. For instance, each SD copy must propagate over either one link (two-party case) or two links (three-party case). For this reason, the three-party architecture (single SCM) can prove more vulnerable to communication failures (two links must be operational). This suggests that the two-party architecture will be more effective under severe failures, and our results support this. On the other hand, the three-party architecture allows replication of SCMs, which provides a greater number of paths through which information can propagate. This suggests (and our results agree) that the three-party architecture with dual SCM provides superior effectiveness over the single-SCM, three-party architecture. Our results also indicate that the dual-SCM three-party architecture yields effectiveness close to that of the two-party architecture.

Regarding consistency-maintenance mechanism, we conclude that polling, with its built-in persistence, should lead to better effectiveness than notification, where events are issued only once with no further action by the sender in response to a REX. Our results support this analysis for the two-party architecture and for the three-party architecture with a single SCM. However, notification appears slightly more effective than polling for the three-party architecture with dual SCM. We suspect this may be because notifications require only that the SCM-to-SU link be operational, while polling also requires the SU-to-SCM link.

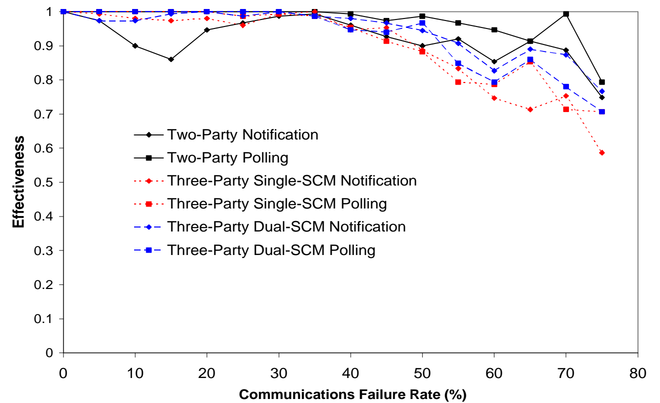

Figure 3. Effectiveness for various combinations of architecture, topology, and consistency-maintenance mechanism, as failure rate increases.

4. DISSECTING RECOVERY STRATEGIES

To further dissect recovery strategies, we decided to factor recovery techniques into four cases: 1) no recovery, 2) soft state only, 3) application persistence only, and 4) both soft state and application persistence.[1] We believe that this finer degree of factoring will enable us to quantify the contribution of various self-healing strategies to overall system effectiveness. Further, we expect that such factoring might reveal interactions among self-healing strategies, and help to identify situations where strategies are redundant, complementary, or conflicting. To explore these

[1] When a failure recovery technique is factored out of an experiment, the related failure detection technique (see Table 1) is also factored out. Eliminating soft state implies that the related heartbeat is ignored, while eliminating application-level persistence implies that the related REX (after bounded retries) is ignored.

ideas, we applied our approach to investigate the contribution of recovery techniques, given various architecture-topology combinations, in the case of one consistency-maintenance mechanism (notification) and one fault type (communication failure).

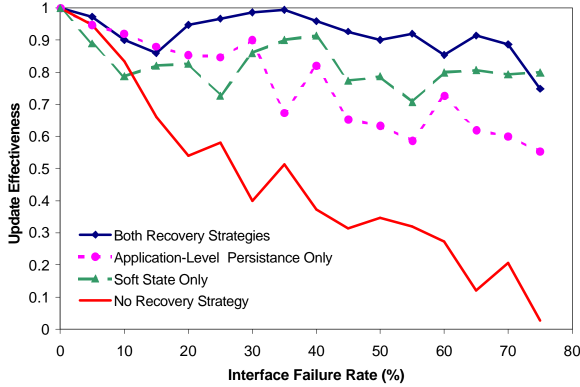

Figure 4. Update effectiveness of two-party notification with soft state, application persistence, and no recovery shown separately.

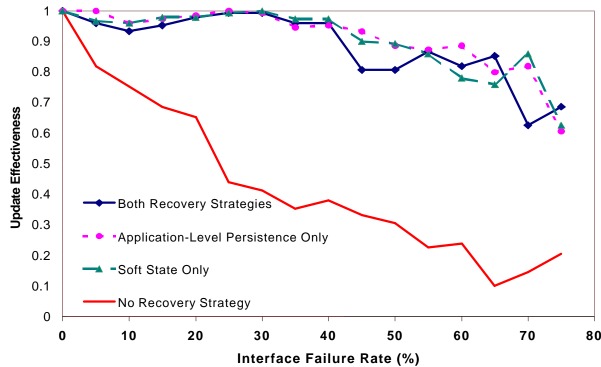

Figure 5. Update effectiveness of three-party (single SCM) notification with soft state, application persistence, and no recovery shown separately.

Figure 4 shows effectiveness for two-party notification as communication failure increases to 75%. The curve representing the use of all recovery techniques was taken from Figure 3. The remaining three curves in Figure 4 depict effectiveness when selected recovery techniques are disabled. Where no recovery is employed, effectiveness decreases nearly linearly as failure rate increases, dropping below 10% when the failure rate reaches 75%. When soft-state recovery is enabled alone, effectiveness improves significantly. Similarly, when application-persistence is enabled alone, effectiveness also improves significantly. Further, Figure 4 shows that application-persistence contributes more to system effectiveness at lower failure rates (30% and below), while soft-state recovery contributes more at higher failure rates. For two-party notification, under communication failure, the two recovery techniques appear complementary.

Figures 5 and 6, which show the contribution of recovery techniques for three-party, single-SCM notification and three-party, dual-SCM notification, yield a different picture. Where all recovery techniques are disabled, effectiveness decreases nearly linearly as failure rate increases; however, the rate of decrease of the three-party dual-SCM architecture appears lower than for the two-party architecture, and effectiveness stays above 10% at the 75% failure rate.

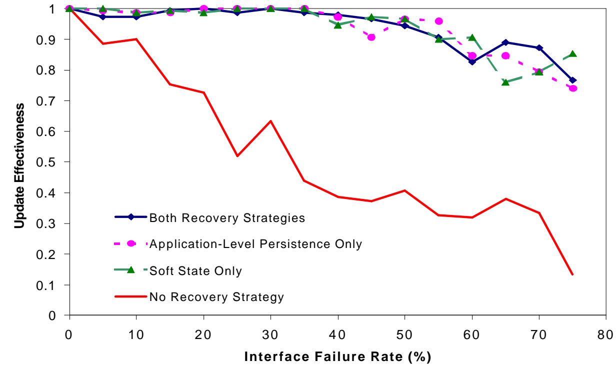

Figure 6. Update effectiveness of three-party notification (dual SCM) with soft state, application-recovery, and no recovery shown separately.

This suggests that increased robustness from a dual-SCM topology slightly mitigates the effects of communication failures. The three-party, single-SCM architecture with no recovery provides the poorest level of performance, reflecting the need to propagate the notification across two links without the alternative path provided by the second SCM. Note, however, that once either recovery technique is enabled in both variants of the three-party architecture, effectiveness improves to the level observed when both recovery techniques are enabled. This result indicates that, for three-party, single and dual-SCM notification, the two recovery techniques (soft state and application persistence) are redundant. These results shown in figures 4 through 6 are summarized in computed summary statistics in Table 2.

	Both Recovery Strategies	Application Persistence Only	Soft-State Only	No Recovery Strategy
Two-Party Notification	**0.921**	**0.763**	**0.824**	**0.466**
Three-Party Notification (Single SCM)	**0.914**	**0.914**	**0.907**	**0.441**
Three-Party Notification (Dual SCM)	**0.942**	**0.938**	**0.942**	**0.533**

Table 2. Summary statistics (mean across all interface failure rates) computed for each curve in the graphs shown in Figure 4 through Figure 6.

5. CONCLUSIONS

Our preliminary results (in Figs. 4-6) show the desirability and feasibility of dissecting the quantitative contributions to system effectiveness of various recovery strategies. Further, our results show that interactions (such as redundancy and complementarity)

between various recovery techniques can be identified and quantified.

Emerging service-discovery protocols provide the foundation for software components to discover each other, to organize themselves into a system, and to adapt to changes in system topology. These capabilities can also be used to effect self-healing in distributed component systems. In this paper, we used architectural models to characterize how architecture, topology, consistency-maintenance mechanism, and failure-recovery strategy each contribute to self-healing during communication failure. Further, in the context of communication failure and using notification as a consistency-maintenance mechanism, we dissected the self-healing properties attributable to recovery techniques and to topology. Our results suggest that it should prove feasible to quantify the ability of individual self-healing strategies to overcome various types of failure. A full understanding of the interactions among self-healing strategies would provide designers of distributed systems with the knowledge necessary to build the most effective self-healing systems with minimum overhead.

6. ACKNOWLEDGMENTS

The work discussed in this paper was funded in part by DARPA, under the auspices of the FTN and DASADA programs.

7. REFERENCES

[1] G. Bieber and J. Carpenter, "Openwings A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems," on the http://www.openwings.org web site.

[2] Ken Arnold et al, The Jini Specification, V1.0 Addison-Wesley 1999. Latest version is 1.1 available from Sun.

[3] Universal Plug and Play Device Architecture, Version 1.0, Microsoft, June 8, 2000.

[4] Luckham, D. "Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events," http://anna.stanford.edu/rapide, August 1996.

[5] Dabrowski, C. and Mills, K., "Analyzing Properties and Behavior of Service Discovery Protocols Using an Architecture-Based Approach", *Proceedings of Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December 2001.

[6] Dabrowski, C. Mills, K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures During Communications Failure", Accepted at *Third Annual Workshop on Software Performance*, Rome Italy, July 2002.

[7] Dabrowski, C. Mills, K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures In Response to Message Loss", Accepted at *Fourth Annual International Workshop on Active Middleware Services*, Edinburgh, Scotland, July 2002.