Understanding Consistency Maintenance in Service Discovery Architectures during Communication Failure

Christopher Dabrowski
NIST
NN Room 560
Gaithersburg, Maryland USA 20899
1-301-975-3249
cdabrowski@nist.gov

Kevin Mills
NIST
NN Room 445
Gaithersburg, Maryland USA 20899
1-301-975-3618
kmills@nist.gov

Jesse Elder
NIST
NN Room 579
Gaithersburg, Maryland USA 20899
1-301-975-4411
jelder@nist.gov

ABSTRACT

Current trends suggest future software systems will comprise collections of components that combine and recombine dynamically in reaction to changing conditions. Service-discovery protocols, which enable software components to locate available software services and to adapt to changing system topology, provide one foundation for such dynamic behavior. Emerging discovery protocols specify alternative architectures and behaviors, which motivate a rigorous investigation of the properties underlying their designs. Here, we assess the ability of selected designs for service-discovery protocols to maintain consistency in a distributed system during catastrophic communication failure. We use an architecture description language, called Rapide, to model two different architectures (two-party and three-party) and two different consistency-maintenance mechanisms (polling and notification). We use our models to investigate performance differences among combinations of architecture and consistency-maintenance mechanism as interface-failure rate increases. We measure system performance along three dimensions: (1) update responsiveness (How much latency is required to propagate changes?), (2) update effectiveness (What is the probability that a node receives a change?), and (3) update efficiency (How many messages must be sent to propagate a change throughout the topology?). We use Rapide to understand how failure-recovery strategies contribute to differences in performance. We also recommend improvements to architecture description languages.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications – *methodologies and tools.*
D.2.5 [**Software Engineering**]: Testing and debugging – *symbolic execution and tracing.*
D.2.8 [**Software Engineering**]: Metrics – *performance measures.*

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. Such environments demand new analysis approaches and tools to design and test software.

In this paper, we use architectural models to assess the ability of selected designs for service-discovery protocols to maintain consistency in a distributed system during catastrophic communication failure. Using an architecture description language (ADL), we model two different architectures (two-party and three-party) and two different consistency-maintenance mechanisms (polling and notification). To provide our models with realistic behaviors, we incorporate consistency-maintenance mechanisms adapted from two specifications: Jini™ Networking Technology[1] [2] and Universal Plug-and-Play (UPnP) [3]. We use our models to investigate performance differences among combinations of architecture and consistency-maintenance mechanism as interface-failure rate increases. We measure system performance along three dimensions: (1) update responsiveness (How much latency is required to propagate changes?), (2) update effectiveness (What is the probability that a node receives a change?), and (3) update efficiency (How many messages must be sent to propagate a change throughout the topology?).

Our modeling and analysis approach builds on earlier work [4] where we derived benefits by creating dynamic models from specifications for service-discovery protocols. Dynamic models

[1] Certain commercial products or company names are identified in this paper to describe our study adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor to imply that the products or names identified are necessarily the best available for the purpose.

enable us to understand collective behavior among distributed components, and to detect ambiguities, inconsistencies and omissions in specifications. In this paper, we apply the same method: (1) construct an architectural model of each discovery protocol, (2) identify and specify relevant consistency conditions that each model should satisfy, (3) define appropriate metrics for comparing the behavior of each model, (4) construct relevant scenarios to exercise the models and to probe for violations of consistency conditions, and (5) compare results from executing similar scenarios against each model. To implement the method, we rely on Rapide [5], an ADL developed at Stanford University. Rapide represents behavior in a form suitable to investigate distributed systems, and comes with an accompanying suite of analysis tools that can execute a specification and can record and visualize system behavior. In this paper, we use Rapide to understand how failure-recovery strategies contribute to differences in performance. Based on our experiences, we also recommend improvements to architecture description languages.

The remainder of the paper is organized in six sections. We begin, in Section 2, by introducing service-discovery protocols and architectures, including a description of procedures to maintain consistency in replicated information. Section 2 also discusses various failures that can interfere with consistency maintenance. In Section 3, we outline some techniques, included in our models, to recover from failures. Section 4 defines an experiment, and related metrics, to compare the performance and overhead exhibited by selected pairings of architecture and consistency-maintenance mechanism while attempting to propagate changes during interface failures. In Section 5, we present results from the experiment, and we discuss causes underlying some of the results. In Section 6, we outline future work to evaluate service-discovery architectures and protocols during message loss and node failure. We conclude in Section 7.

2. SERVICE DISCOVERY SYSTEMS

Service-discovery protocols enable software components in a network to discover each other, and to determine if discovered components meet specific requirements. Further, discovery protocols include *consistency-maintenance mechanisms*, which can be used by applications to detect changes in component availability and status, and to maintain, within some time bounds, a consistent view of components in a network. Many diverse industry activities explore different approaches to meet such requirements, leading to a variety of proposed designs for service-discovery protocols [2, 3, 6-14]. Some industry groups approach the problem from a vertically integrated perspective, coupled with a narrow application focus. Other industry groups propose more widely applicable solutions. For example, a team of researchers and engineers at Sun Microsystems designed Jini Networking Technology [2], a general service-discovery mechanism atop Java^{TM}, which provides a base of portable software technology. As another example, a group of engineers at Microsoft and Intel conceived Universal Plug-and-Play [3] in an attempt to extend plug-and-play, an automatic intra-computer device-discovery and configuration protocol, to distributed systems. The proliferation of service discovery protocols motivates deeper analyses of their designs.

To help us compare designs, we developed a general structural model, documented using the UML (Unified Modeling Language). Our general model provides a basis for comparative analysis of various discovery systems by representing the major architectural components with a consistent and neutral terminology (see first column in Table 1). The main components in our general model include: (1) service user (SU), (2) service manager (SM), and (3) service cache manager (SCM), where the SCM is an optional element not supported by all discovery protocols. These components participate in the discovery, information-propagation, and consistency-maintenance processes that comprise discovery protocols. A SM maintains a database of service descriptions, (SDs), each SD encoding the essential characteristics of a particular service or device (Service Provider, or SP). Each SD contains the identity, type, and attributes that characterize a SP. Each SD also includes up to two software interfaces (an application-programming interface and a graphic-user interface) to access a 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 requirements provided by SUs. Table 1 shows how these general concepts map to specific concepts from Jini, UPnP, and the Service Location Protocol (SLP) [8]. The behaviors by which SUs discover and maintain consistency in desired SDs depend partly upon the service-discovery architecture employed.

Table 1. Mapping concepts among service-discovery systems.

Generic Model	Jini	UPnP	SLP
Service User (SU)	Client	Control Point	User Agent
Service Manager (SM)	Service or Device Proxy	Root Device	Service Agent
Service Provider (SP)	Service	Device or Service	Service
Service Description (SD)	Service Item	Device/Service Description	Service Registration
Identity	Service ID	Universal Unique ID	Service URL
Type	Service Type	Device/Service Type	Service Type
Attributes	Attribute Set	Device/Service Schema	Service Attributes
User Interface	Service Applet	Presentation URL	Template URL
Program Interface	Service Proxy	Control/Event URL	Template URL
Service Cache Manager (SCM)	Lookup Service	not applicable	Directory Service Agent (optional)

2.1 Alternative Architectures

Broadly speaking, system architecture comprises a set of components, and the connections among them, along with the relationships and interactions among the components. In our application, we represent the architecture of a discovery system using an architectural model, which expresses structure (as components, connections, and relations), interfaces (as messages received by components), behavior (as actions taken in response to messages received, including generation of new messages), and consistency conditions (as Boolean relations among state variables maintained across different components). Our initial analysis of six distinct discovery systems revealed that most designs use one of two underlying architectures: two-party and three-party.

2.1.1 Two-Party Architectures

A two-party architecture consists of two major components: SMs and SUs. In this study, we use a two-party architecture arranged in a simple topology consisting of one SM and five SUs, as depicted in Figure 1. To animate the architecture, we chose behaviors for discovery, information propagation, and consistency maintenance, as described in the specification for UPnP. Upon startup, each SU and SM engages in a discovery process to locate 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, used to send messages from a source to a group of receivers. Upon receiving these announcements, SUs with matching requirements use a HTTP/TCP (HyperText Transfer Protocol/transmission-control protocol) unicast link (for message exchanges between two specific parties) to request, directly from the SM, copies of the SDs associated with relevant SPs. The SU stores SD copies in a local cache. Alternatively, the SU may engage in an aggressive-discovery process, where the SU transmits SD requirements, as *Msearch* queries, on the UPnP multicast group. Any SM holding a SD with matching requirements may use a HTTP/UDP (user-datagram protocol) unicast link to respond (after a jitter delay) directly to the SU. Whenever a UPnP SM responds to an *Msearch* query (or announces itself), it does so with a train of $(3 + 2d + k)$ messages, where d is the number of distinct devices and k is the number of unique service types managed by the SM. For each appropriate response, the SU uses a HTTP/TCP unicast link to request a copy of the relevant SDs, caching them locally.

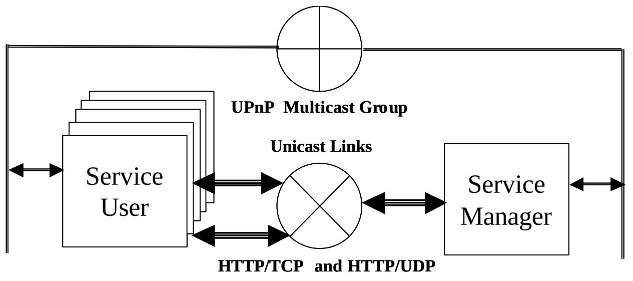


Figure 1. Two-party service-discovery architecture deployed in a six-node topology: five service users (SUs) and one service manager (SM).

To maintain a SD in its local cache, a SU expects to receive periodic announcements from the relevant SM. In UPnP, the SM announces the existence of SDs at a specified interval, known as a Time-to-Live, or TTL. Each announcement specifies the TTL value. If the SU does not receive an announcement from the SM within the TTL (or a periodic SU *Msearch* does not succeed within that time), the SU may discard the discovered SD. We selected the minimum TTL of 1800 s, as recommended by the UPnP specification. (See Tables 2 and 4 for a summary of relevant parameter values used in this paper.)

2.1.2 *Three-Party Architectures*

A three-party architecture consists of SMs, SUs, and SCMs, where the number of SCMs represents a key variable. In this study, we model a three-party architecture with one SM and five SUs, as shown in Figure 2. We anticipate that under failure conditions, increasing the number of SCMs will increase the chance of successful rendezvous among components, leading to better propagation of information updates from SMs to SUs. To investigate this, we vary the number of SCMs in our three-party architectural model. To animate our three-party model, we choose 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. Elsewhere [4], we describe these procedures in detail. Here, we simply summarize. Upon initiation, a Jini component enters aggressive discovery, where it transmits probes on the aggressive-discovery multicast group at a fixed interval (5 s recommended) for a specified period (seven times recommended), or until it has discovered a sufficient number of SCMs. Upon cessation of aggressive discovery, a component enters lazy discovery, where it listens on the lazy-discovery multicast group for announcements sent at intervals (120 s recommended) by SCMs. Our three-party model implements both the aggressive and lazy forms of Jini multicast discovery.

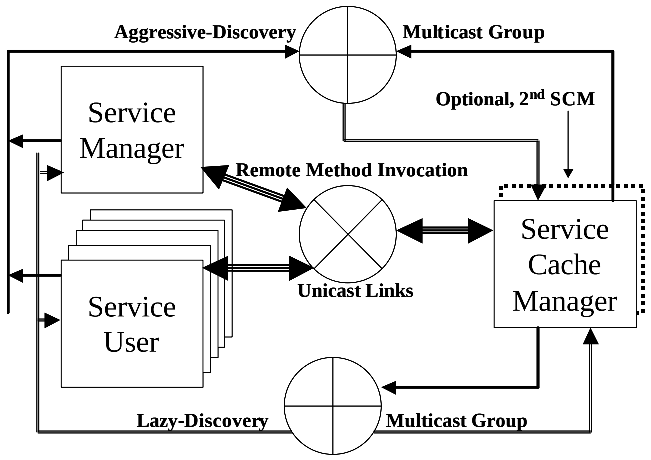


Figure 2. Three-party service-discovery architecture deployed in a seven- or eight-node topology: five service users (SUs), a service manager (SM), and a service cache manager (SCM), with an optional 2nd SCM.

Once discovery occurs, a SM deposits a copy of the SD for each of its services on the discovered SCM. The SCM caches this deposited state, but only for a specified length of time, or TTL. To maintain a SD on the SCM beyond the TTL, a SM must refresh the SD. In this way, if the SM fails, then the SCM can purge any SDs deposited by the SM. To make behavior as consistent as possible across our models for both the two-party and three-party architectures, we selected 1800 s as TTL for a SD to be cached by a SCM. Using these techniques, SUs and SPs rendezvous through SDs registered by SMs with particular SCMs, where the SCMs are found through a discovery process. The SCMs match SDs provided by SMs to SU requirements, and forward matches to SUs, which then access the appropriate SPs.

2.2 Consistency Maintenance Mechanisms

After initial discovery and information propagation (through SDs), service-discovery protocols provide consistency-maintenance mechanisms that applications can use to ensure that changes to critical information propagate throughout the system. Critical information may consist of service availability and capacity, or updates to descriptive information about service capabilities, which may be necessary for a SU to effectively use a discovered service. In our study, we consider two basic consistency-maintenance mechanisms, polling and notification, along with accompanying mechanisms to propagate new information.

2.2.1 *Polling*

In polling, a SU periodically sends queries to obtain up-to-date information about a SD that was previously discovered, retrieved, and cached locally. In a two-party architecture, the SU issues the query directly to the SM from which the SD was obtained. In this study, we use the UPnP *HTTP Get* request mechanism to poll the SM to retrieve a SD associated with a specific URL (uniform-resource locator). In response, the SM provides a SD containing a list of all supported services, including their relevant attributes.

Polling in a three-party architecture consists of two independent processes. In one process, a SM sends a *ChangeService* request to propagate an updated SD to each SCM where the SD was originally cached. In the second process, each SU polls relevant SCMs by periodically issuing a *FindService* request, effectively a query with a set of desired SD requirements. The SCM replies with a *MatchFound* that contains the relevant information for any matching SDs. In our study, we adopt a 180-second interval for polling in both architectures.

2.2.2 *Notification*

In notification, immediately after an update occurs, a SM sends events that announce a SD has changed. To receive events about a SD of interest, a SU must first register for this purpose. In the two-party architecture, the SU registers directly with a SM. We model this procedure using the UPnP event-subscription mechanism, where the SU sends a *Subscribe* request, and the SM responds by either accepting the subscription, or denying the request. The subscription, if accepted, is retained for a TTL, which may be refreshed with subsequent *Subscribe* requests from the SU. In our experiment, we chose 1800 s as TTL for event subscriptions in both architectures.

In a three-party architecture, a SU registers with a SCM to receive events using a procedure analogous to that used by a SM to propagate a SD. As with SD propagation, the SCM grants event registrations for a TTL, which may be refreshed. When a SD update occurs, the SM first issues a *ChangeService* request to all SCMs to which it originally propagated the SD. The SCM then issues a *MatchFound* to propagate the event to all SUs that have registered to receive events about the SD.

2.3 The Nature and Import of Failures

The foregoing discussion, while oversimplified, highlights the complexity inherent in discovery protocols. Additional complexity arises from uncertainty, as nodes, processes, and links can appear and disappear without warning. Discovery protocols must include behavior to cope with such changes. In this section, we address the nature of various failures that can arise, and we consider the implication of such failures on the behavior of discovery protocols, and on the application software that depends upon them.

2.3.1 *Classifying Failures*

In our research, we focus particularly on failures that can exist within a hostile environment, such as encountered during military or emergency-response operations. We can classify such failures in two general categories: (1) communication failures and (2) process failures. Communication failures can arise due to enemy jamming, or other interference, due to congestion, due to physical severing of cables, due to improperly configured or sabotaged routing tables, or due to multi-path fading as nodes move across a terrain. We can subdivide communication failures into three classes: interface failures, message loss, and path failures. This paper considers only interface failure. A communication interface in a node 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. Message loss, a less severe failure, implies that individual messages may be lost, either sporadically or in bursts. Path loss appears as a blocked communication route between two nodes, or areas, in the network. A path can be blocked in one or both directions.

Process failures can be caused by enemy bombardments or cyber attacks, by programming errors, or by hardware failures. We can subdivide process failures into node and thread failures. During a catastrophic failure, processing in a node ceases, and the node must reinitialize before processing resumes. Some information maintained by the node may persist across the failure, while other information may be lost. The nature and condition of persistent information could prove crucial to a node's behavior after processing resumes. Of course, the node might never reappear. Thread failures, while less catastrophic, can be more troublesome than node failures. A node might rely on certain long-running threads to react to events from other nodes. Failure of selected threads can interfere with the operation of the node, as well as other nodes in a distributed system. In some cases, a node can appear to be present, while being effectively inoperable.

2.3.2 *Failure Recovery in Service Discovery Systems*

In service-discovery systems, failure-recovery responsibilities are divided among three parties: (1) lower-layer protocols, (2) discovery protocols, and (3) applications. Discovery protocols and applications use the services of three classes of lower-layer protocols: (1) unreliable unicast protocols, (2) unreliable multicast protocols, and (3) reliable unicast protocols. Unreliable protocols, whether unicast or multicast, neither recover nor signal lost messages; thus, neither source nor destination will learn of a loss. Further, multicast protocols exchange messages along a tree of receivers. For this reason, a multicast message might be received by some nodes, but not by others. A failure near a multicast source prevents messages from being received by any node in the multicast tree, while a failure near a receiver prevents messages from being received by only a single node in the multicast tree. Of course, failures at intermediate points in the multicast tree could result in messages being lost to subsets of receivers. Since unreliable protocols provide no guarantees, recovery must be provided by mechanisms at a higher layer.

Reliable unicast protocols include mechanisms that attempt to ensure delivery of messages by detecting and retransmitting lost messages. Of course, the reliability schemes may eventually give up if too many retransmissions are needed (which might indicate node, interface, or path failure). In such cases, the reliable unicast protocol will signal to a higher layer that a message could not be delivered. Some ambiguity does exist, however, when using reliable unicast protocols to send request-response message pairs, as is the case for discovery systems. After submitting a request through a reliable unicast protocol, a requesting process might wait for a corresponding response from a remote process. For example, Jini can use Remote Method Invocation (RMI) over TCP to invoke a method on a remote object, and to receive a response. Similarly, UPnP uses TCP to submit HTTP requests and receive HTTP responses. In such cases, the RMI layer or the TCP layer can signal a remote exception (REX). The requesting process cannot determine whether a REX was caused by failure to transmit the request or by failure to receive a response from the remote process. The responding process has more information, as it does not receive a REX when an inbound request fails, but does receive a REX when its outbound response fails. In essence, while reliable unicast protocols attempt to deliver messages in the face of various communication failures, ultimately the reliability mechanisms might prove insufficient, causing a higher-layer process to be notified of the failure. In such cases, the higher-layer process is free to determine an appropriate recovery strategy.

3. MODELING RECOVERY STRATEGIES

Our architectural models incorporate three classes of failure-recovery strategies: (1) recovery by lower-layer protocols, (2) recovery by discovery protocols, and (3) recovery by application software. For each class, we outline the strategies (see Table 2) included in our models.

Table 2. Summary of recovery responsibilities and strategies as implemented within our models for two- and three-party architectures.

Responsible Party	Recovery Mechanism	Two-Party Architecture (UPnP)	Three-Party Architecture (Jini)
Lower-Layer Protocols	UDP	No recovery	No recovery
	TCP	Issue REX in 30-75 s	Issue REX in 30-75 s
Discovery Protocols	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
Application Software	Ignore REX	SU: *HTTP Get* Poll SM: Notification	SU: *FindService* Poll SCM: Notification
	Retry after REX	SU: *HTTP Get* after discovery retry in 180 s (retries ≤ 3) *Subscribe* requests retry in 120s	SM: depositing or refreshing SD copy on SCM retry in 120s SU: registering and refreshing notification requests with 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

3.1 Recovery by Lower-Layer Protocols

Our models operate over two types of channels: unreliable, simulating the UDP in both multicast and unicast forms, and reliable, simulating the TCP. In UDP simulation, we discard messages lost due to transmission errors, and we discard messages lost due to path and interface failures. During path failure, messages can be discarded in one or both directions. During interface failure, we discard all messages sent from a node with a failed transmitter, and we discard all messages inbound for a node with a failed receiver. Neither sender nor receiver learns the fate of lost messages.

In the TCP simulation, our model proves more complex. For messages lost to transmission errors, we schedule a retransmission (roughly within a round-trip time, or RTT). We increase the RTT by about 25% with each successive retransmission. If successive retransmissions exceed a threshold (three in the current study), then we discard the message and issue a REX. For messages lost to interface or path failure, we model TCP connection establishment procedures by discarding the message and waiting for a period, uniformly distributed between an upper and lower bound (30-75 s in the current study), then we signal a REX. When discarding a request, we signal a REX to the requester, but when discarding a response, we signal a REX to both parties.

3.2 Recovery by Discovery Protocols

Discovery protocols include built-in robustness measures to deal with the possibilities of UDP message loss and node failure. Discovery protocols specify periodic transmission of key messages. For example, Jini requires a node to engage in aggressive discovery on startup, and then to enter lazy discovery, where all SCMs periodically announce their presence. In a similar lazy discovery, UPnP requires SMs to periodically announce their presence. While not specifying aggressive discovery, UPnP permits SUs to issue *Msearch* queries at any time. To compensate for the different announcement intervals recommended for Jini and UPnP, we chose to have UPnP SUs issue *Msearch* queries every 120 s, but only after a SU purges a SD from its local cache. Once a SU regains its desired SD, the related *Msearch* queries cease. Whenever a UPnP SM announces itself or responds to an *Msearch* query, it sends *n* copies of each message, where *n* is a retransmission factor (two in the current study) recommended by the UPnP specification to compensate for possible UDP message loss. In both Jini and UPnP, each announcement includes a TTL. Receiving nodes can cache the information in the announcement until the TTL expires; then the information must be purged from the cache. In this way, each node in the system eliminates residual information about failed or unreachable nodes. Our models incorporate these failure-recovery behaviors.

3.3 Recovery by Application Software

When discovery nodes communicate over a reliable channel, a REX may occur. Response to a REX is left to the application. In our models, depending on the situation, we implement three different strategies: (1) ignore the REX, (2) retry the operation for some period, and (3) discard knowledge. The retry strategy attempts to recover from transient failures. The discard strategy, which occurs following repeated failure of the retry strategy, relies upon discovery mechanisms to recover from more persistent failures.

3.3.1 Ignore the Remote Exception

In many cases, we simply ignore a REX. In general, our models ignore a REX received when attempting to respond to a request. A SU can ignore a REX received in response to a poll, *FindService* or *HTTP Get*, because the poll recurs at an interval. The SCM (three-party model) or the SM (two-party model) also ignores a REX received while attempting to issue a notification. This behavior, which is described in both the Jini and UPnP specifications, depends upon reliable lower-layer protocols to provide robustness for notifications. Notifications include sequence numbers that allow a receiving node to determine whether or not previous notifications were missed.

3.3.2 Retry the Operation

In our models, we retry selected operations in the face of a REX. The UPnP specification separates the operation of discovering a resource from obtaining a description of the resource (Jini combines these operations). Without a description, the resource cannot be used. For this reason, in our two-party model, a SU must issue a *HTTP Get* to obtain a description. If no description arrives within 180 s, then our model retries the *HTTP Get*. If unsuccessful after three attempts, the SU ceases the retries, but sets a flag reminding itself to reissue a *HTTP Get* when the resource is next announced. Our three-party model, based on Jini, also contains a retry strategy, but associated with attempts to register or change a SD with a SCM. In these cases, the SM retries a *ChangeService* or *ServiceRegistration* 120 s after receiving a REX. Similarly, when a SU receives a REX (from either a SM or SCM) in response to a request to register for notification, the SU retries the registration in 120 s. These retries occur until some time bounds, after which the SM discards knowledge of the SCM.

3.3.3 Discard Knowledge

Both our two-party and three-party models include the possibility that an application can discard knowledge of previously discovered nodes. In UPnP, after failure to receive announcements from the SM within a TTL, a SU discards a SM and any related SDs. We implement this behavior in our two-party model. 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 when 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, either through lazy discovery (Jini or UPnP announcements) or aggressive discovery (UPnP *Msearch* queries).

4. EXPERIMENT DESIGN AND METRICS

In this paper, we investigate the following question: How do alternative service-discovery architectures, topologies, and consistency-maintenance mechanisms perform under deadline during interface failure? To address this question, we deploy a two-party and three-party architecture (recall Figures 1 and 2), each in a topology that includes one SM and five SUs. In the three-party case, we use two topologies, one with one SCM and another with two SCMs. To establish initial conditions, we exercise each topology until discovery completes, and the initial information (a SD) propagates to all SUs. To begin the experiment, we introduce a change in the SD at the SM, and we establish a deadline, *D*, before which the change must propagate to all SUs. We measure the number of messages exchanged and the latency required to propagate the new information, or until *D*, under two different consistency-maintenance mechanisms: polling and notification. We repeat this experiment while varying the percentage of interface-failure time for each node up to 75% (in increments of 5%). We provide further details below.

Table 3. Experiment combinations.

Architectural Variant	Protocol Basis	Consistency-Maintenance Mechanism
Two-Party	UPnP	Polling
Two-Party	UPnP	Notification (with notification registration on SM)
Three-Party (Single SCM)	Jini	Polling (with service registration on SCM)
Three-Party (Single SCM)	Jini	Notification (with service registration and notification registration on SCM)
Three-Party (Dual SCM)	Jini	Polling (with service registration on SCM)
Three-Party (Dual SCM)	Jini	Notification (with service registration and notification registration on SCM)

4.1 Experiment Combinations

To compare change propagation in two- and three-party architectures, we use our models to combine the architectures with different consistency-maintenance mechanisms. Table 3 depicts the six combinations. Each experiment runs one combination from time zero until *D*, while introducing failures at each node (see 4.3). Each experiment aims to restore consistency among the changed SD held by the SM and the cached copies of the SD held by all of the SUs.

4.2 Tracking Consistency

To track consistency in our experiment, we employ property analysis [4], using a single consistency condition: service attributes for a SD discovered by a SU should have the same values as the attributes of the SD being maintained by the SM that manages the SD. More formally,

```
FOR All (SM, SU, SD)
(SM, SD [Attributes1]) isElementOf SM managed-services   AND
(SM, SD [Attributes2]) isElementOf SU discovered-services
        implies Attributes1 equals Attributes2
```

The condition is incorporated directly into our models and checked using Rapide procedural code. We establish an initial system state in which this condition holds, and then introduce a change in (SM, SD [Attributes1]), which negates the condition for all SUs. Then, we monitor updates to (SM, SD) tuples in the set of discovered-services maintained by individual SU's to determine if the condition becomes true. Note that if a SU discards its (SM, SD) tuple, the tuple must be recovered before the condition can be satisfied. These consistency checks form the basis for our measurements.

4.3 Generating Interface Failures

We set aside an interval, up to time *Q*, to complete initial discovery and information propagation. In our experiments, Q = 100 s and D = 5400 s. We choose a time, randomly distributed on the uniform interval *Q* to *D*/2, to introduce a change into the SD on the SM. We also choose times, randomly distributed on the uniform interval *Q* to [D - (D x F)], for each node to suffer an interface failure, where *F* is the interface-failure rate, which defines the duration of failures as follows. Once activated, each failure remains in effect for a duration of D x F, after which the failure is remedied. We choose interface failures to be of equal and increasing length to give a suitable basis for comparative analysis. When activating each interface failure, we choose with equal likelihood that the transmitter, receiver, or both fail. Table 4 summarizes most of the relevant parameters and values for our experiments.

4.4 A Sample Run

Figure 3 shows partial results from a sample run for the three-party architecture, with two SCMs, using notification as the consistency-maintenance mechanism. In this run, *F* was 0.05, and so each failure occurred between 100 and 5130 s [D - (D x F)], and lasted for 270 s (D x F). Figure 3 shows the time when each interface failed, and recovered. The performance section of the figure lists two times for each node: loss of consistency and restoration of consistency, or *D* where inconsistency remains. The figure also lists two message counts for each node: messages sent to restore consistency and total messages sent. For each SM and SCM, the first count includes messages sent while any SU remains inconsistent. In this sample run, SUs 1, 2, 4, and 5 and both SCMs became consistent quickly, within 0.00109 s, which represents the time necessary to propagate the change from the SM to at least one SCM, match the changed SD registration to all the SU notification requests registered on the SCM, and forward the matches. However SU 3, whose receiver failed at an inopportune time, never heard the notification and continued in an inconsistent state for the remainder of the run. This illustrates how lack of robustness in the notification mechanism can lead to prolonged inconsistent states.

Table 4. Values for relevant parameters.

	Parameter	Value
Behavior in both two- and three-party architectures	Polling interval	180 s
	Registration TTL	1800 s
	Time to retry after REX (if applicable)	120 s
UPnP-specific behavior for two-party architecture	Announce interval	1800 s
	Msearch query interval	120 s
	SU purges SD	At TTL expiration
Jini-specific behavior for three-party architecture	Probe interval	5 s (7 times)
	Announce interval	120 s
	SM or SU purges SCM	After 540 s with only REX
Interface failure parameters	Failure incidence	Once per run for each node
	Failure scope	Transmitter, receiver, or both with equal likelihood
	Failure duration	5% increments of 5400 s from 0 to 75%
Transmission and processing delays	UDP transmission delay	10 us constant
	TCP transmission delay	10-100 us uniform
	Per-item processing delay	100 us for cache items 10 us for other items

```
Rate - 5
Run number - 21

SM 1  OUT Interface       down 365, up 635

SCM 1 OUT Interface       down 2417, up 2687
SCM 2 IN & OUT Interface  down 519, up 789

SU 1  IN Interface        down 2238, up 2508
SU 2  IN Interface        down 3256, up 3526
SU 3  IN Interface        down 207,  up 477
SU 4  OUT Interface       down 2876, up 3146
SU 5  IN Interface        down 4478, up 4748

Performance:

  SM  1 346.00000 346.00000 6 17
  SCM 1 346.00000 346.00016 61 102
  SCM 2 346.00000 346.00015 61 105
  SU  1 346.00000 346.00109 0 11
  SU  2 346.00000 346.00109 0 11
  SU  3 346.00000 5400.00000 4 11
  SU  4 346.00000 346.00109 0 11
  SU  5 346.00000 346.00114 0 11
```

Figure 3. Console output from a sample run: three-party, two SCMs, notification, F = 5%, Q =100 s, and D = 5400 s.

4.5 Metrics

We use the data collected from experiment runs to compute three metrics: update responsiveness, update effectiveness, and update efficiency. We define these below.

4.5.1 Update Responsiveness

Assuming information is created at a particular time and must be propagated by a deadline, then the difference between the deadline and the creation time represents available time in which to propagate the information. Update Responsiveness, *R,* measures the proportion of the *available time remaining* after the information is propagated. More formally, let *D* be a deadline by which we wish to propagate information to each SU-node *n* in a service discovery topology. Let t_C be the creation time of the information that we wish to propagate, where $t_C < D$. Let $t_{U(n)}$ be the time that the information is propagated to SU *n*, where *n* = 1 to *N*, and *N* is the total number of SUs in a topology. Define change-propagation latency (*L*) for SU *n* as: $L_n = (t_{U(n)} - t_C)/(\max(D,\ t_{U(n)}) - t_C)$. This is effectively the proportion of available time used to propagate the change to SU *n*. The numerator represents the time at which the SU achieved consistency after the update occurred. The denominator represents the time available to propagate the change. The term $\max(D, t_{U(n)})$ accounts for cases where $t_{U(n)} > D$. Define *R* for SU *n* as: $R_n = 1 - L_n$. R_n is the proportion of *available time remaining* after propagating a change to SU *n*.

4.5.2 Update Effectiveness

Update Effectiveness, *U*, measures the probability that a change will propagate successfully for a given SU, i.e., $t_{U(n)} < D$. More formally, assuming definitions from 4.5.1 hold, let *X* be the number of runs (30 here) during which a particular topology is observed under identical conditions. Recalling that *N* is the total number of SUs in a topology, define the number of SUs observed under identical conditions as: $O = X \cdot N$. Define *U*, the probability that $t_{U(n)} < D$, as: $U = 1 - P(F)$, where $P(F) = (\Sigma i \Sigma j$ (one if $R_{i,j}$ equals 0 and zero otherwise))/O and where $i = 1...X$ and $j = 1...N$.

4.5.3 Update Efficiency

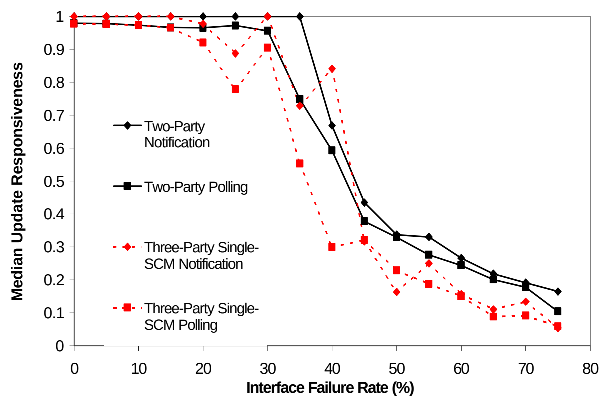
Given a specific service-discovery topology, examination of the available architectures (two-party and three-party) and consistency-maintenance mechanisms (polling and notification) reveals a minimum number of messages, *M*, that must be sent to propagate a change to all SUs. In our topology, *M* (*M* = 7) occurs when using notification to propagate information in a three-party architecture with one SCM. Update Efficiency, *E*, can be defined as the ratio of *M* to the actual number of messages observed. More formally, let *S* be the number of messages sent while attempting to propagate a change from a SM to SUs in a given run. Define average *E* as: $E_{avg} = (\Sigma k(M/S_k))/X$, where $k = 1..X$.

5. RESULTS AND DISCUSSION

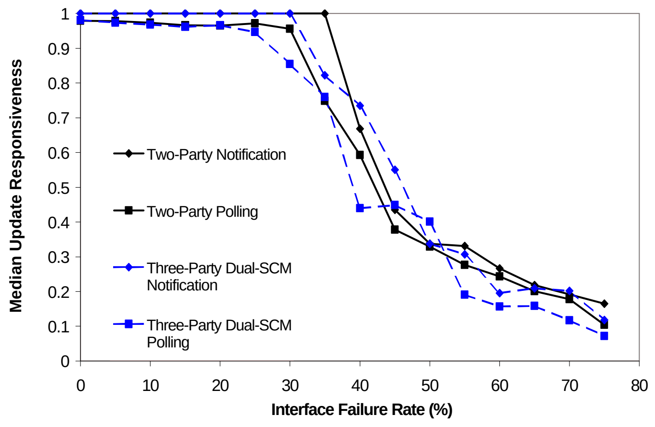
In this section, after showing results from our experiments, we consider the relative performance of our models. We propose reasons for performance differences, subject to further analysis and verification by on-going research. We also use Rapide to examine selected saw-tooth behaviors, and we outline suggestions for improving ADLs (based on our experiences with Rapide).

5.1 Results

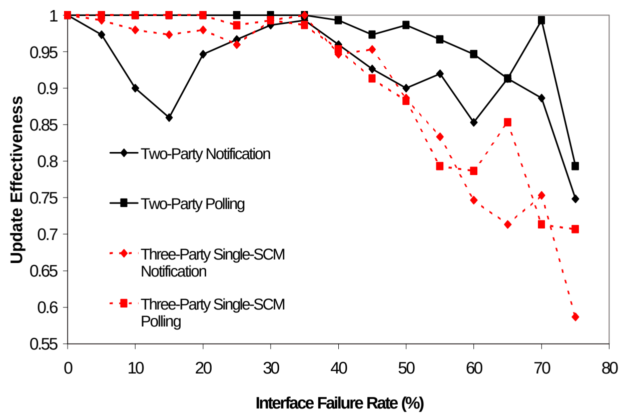
In a series of six graphs, which have identical abscissas (interface-failure rate, increasing from 0% to 75% in increments of 5%) and ordinates (one of the three metrics ranging between 0 and 1), we plot selected measurements generated from our models. Each graph compares four of the configurations in Table 3 against one of the metrics: update responsiveness (median), effectiveness, or efficiency (average). We choose the median as a measure of update responsiveness because the measured data tend to clump in distinct concentrations. Averages proved less representative of the data. Figure 4(a) compares responsiveness from our two-party model against that from our single-SCM, three-party model, for both polling and notification. Figure 4(b) provides a similar comparison, but substitutes the results from our dual-SCM, three-party model in place of results from our one-SCM, three-party model. Figures 4(c) and 4(d) compare update effectiveness using



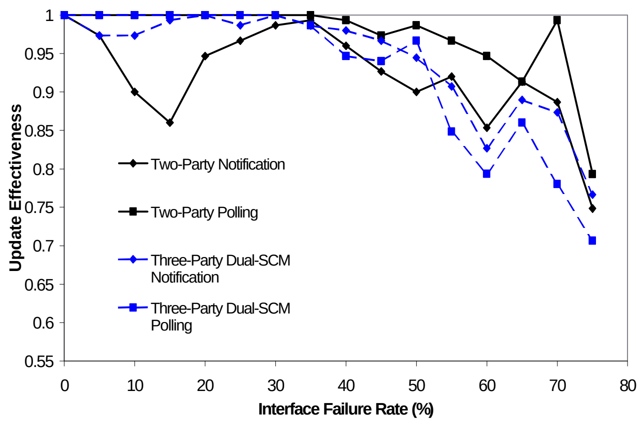
(a) Median Update Responsiveness of Two-Party vs. Three-Party (Single-SCM)



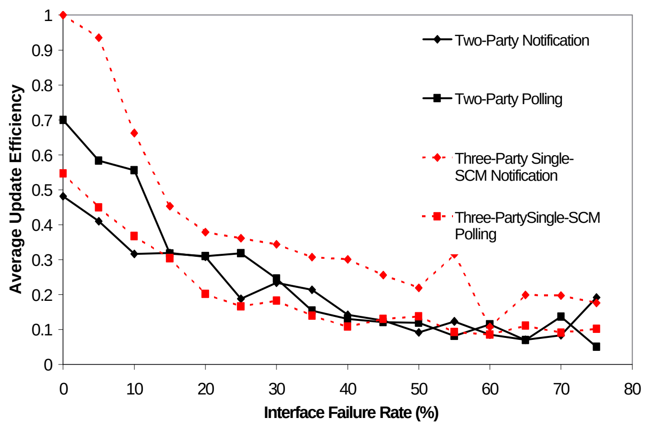
(b) Median Update Responsiveness of Two-Party vs. Three-Party (Dual-SCM)



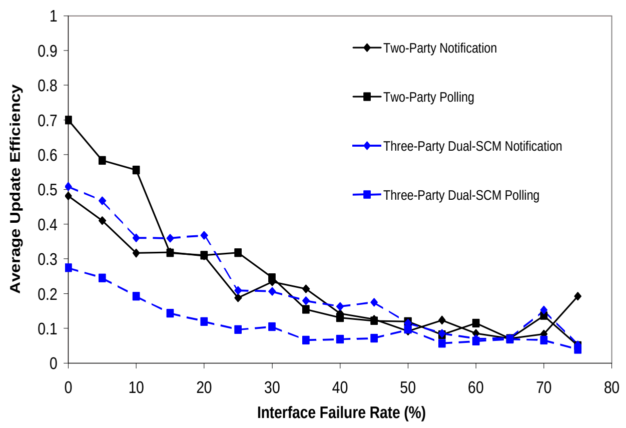
(c) Update Effectiveness of Two-Party vs. Three-Party (Single-SCM)



(d) Update Effectiveness of Two-Party vs. Three-Party (Dual-SCM)



(e) Update Efficiency of Two-Party vs. Three-Party (Single-SCM)



(f) Update Efficiency of Two-Party vs. Three-Party (Dual-SCM)

Figure. 4. Graphs comparing combinations of architecture, topology, and consistency-maintenance mechanism.

the same combinations. Figures 4(e) and 4(f) use the same combinations, but compare update efficiency. The graphs reporting measures of responsiveness and effectiveness depict a system undergoing a phase-transition from peak performance (where changes propagate quickly) to non-performance (where changes fail to propagate). Regarding efficiency, the graphs show a system that begins at its best efficiency (without interfering failures) and then asymptotically approaches zero efficiency as the failure rate increases toward 100%. The graphs (particularly those showing update effectiveness) also depict several eccentricities, in the form of saw-tooth behaviors. Using the analysis and visualization tools provided by Rapide, we were able to investigate the causes underlying these eccentricities (see 5.3). Because the graphs can be difficult to interpret, we compute summary statistics (see Table 5) for each of our six combinations. Each summary statistic reflects the mean of a particular metric, when averaged across all interface-failure rates, for a specified configuration. To indicate the uncertainty associated with our measurements, we also give (see Table 6) the upper and lower bounds (computed using an appropriate standard error formula for each metric) associated with selected interface-failure rates (5%, 40%, and 75%) for each of our curves.

Table 5. Summary statistics (mean across all interface-failure rates) computed for each curve given in the graphs shown in Figures 4(a) through 4(f).

	Mean (across all interface-failure rates)		
	Median Responsiveness	Effectiveness	Average Efficiency
Two-Party Notification	0.663	0.921	0.212
Two-Party Polling	0.615	0.973	0.251
Three-Party Notification (Single SCM)	0.601	0.894	0.389
Three-Party Polling (Single SCM)	0.530	0.911	0.201
Three-Party Notification (Dual SCM)	0.655	0.942	0.221
Three-Party Polling (Dual SCM)	0.587	0.927	0.110

5.2 Understanding Relative Performance

Below, we discuss the results for each of our three metrics. The reader should note that engineering trade-offs exist among these metrics: responsiveness, effectiveness, and efficiency.

5.2.1 Responsiveness

Results in Figs. 4(a) and 4(b) and the first column of Table 5 show that the various combinations of architecture and behavior exhibit similar responsiveness, where the mean median ranges between 0.663 and 0.530. Table 6, which reports uncertainty in the results, confirms a rough similarity in responsiveness. Similarity arises because interface failures interfere with both polling and notification, requiring nodes to rely on recovery mechanisms in the underlying discovery protocols to restore consistency. Absent failures, notification proves more responsive because change notices are issued to interested parties immediately after a change occurs, while polling incurs some lag time. The presence of interface failures complicates the situation. First, if a required interface is not operating when a notification is issued, then an update will be lost. Second, when polls fail for an extended period (likely during high interface-failure rates), then polling ceases and updates can be missed. Under both (polling and notification) mechanisms, restoring consistency depends upon the recovery mechanisms in the discovery protocol.

Table 6. Depicts upper and lower bounds of the 95% C.I., computed using appropriate statistical techniques, for each metric and all experiment combinations at selected interface-failure rates.

	Responsiveness			Effectiveness			Efficiency		
	5%	40%	75%	5%	40%.	75%	5%	40%	75%
Two-Party Notification	1.000 1.000	0.561 0.783	0.111 0.162	0.970 0.977	0.954 0.966	0.709 0.787	0.354 0.467	0.065 0.220	0.031 0.354
Two-Party Polling	0.975 0.980	0.501 0.849	0.076 0.138	1.000 1.000	0.993 0.993	0.760 0.826	0.501 0.666	0.031 0.230	0.042 0.059
Three-Party Notification (Single SCM)	1.000 1.000	0.605 1.000	0.042 0.095	0.993 0.993	0.939 0.955	0.521 0.652	0.827 1.000	0.099 0.504	0.033 0.320
Three-Party Polling (Single SCM)	0.974 0.980	0.244 0.412	0.043 0.083	1.000 1.000	0.946 0.960	0.660 0.753	0.387 0.512	0.043 0.173	0.040 0.164
Three-Party Notification (Dual SCM)	1.000 1.000	0.562 1.000	0.099 0.143	0.970 0.977	0.977 0.983	0.730 0.803	0.335 0.599	0.035 0.290	0.009 0.096
Three-Party Polling (Dual SCM)	0.974 0.986	0.391 0.543	0.056 0.096	1.000 1.000	0.939 0.955	0.660 0.753	0.218 0.273	0.033 0.103	0.019 0.059

The recovery mechanisms, as implemented in our models, exhibit similar responsiveness: rediscovery of lost nodes will occur within 120 s after restoration of a failed interface. In the three-party case, periodic (120 s) announcements by each SCM (lazy-discovery procedures) ensure rediscovery. Similarly, in the two-party model, the periodic (120 s) *Msearch* queries by each SU (aggressive-discovery procedures) also ensure rediscovery. In this way, restoration of a failed interface leads to rediscovery of lost nodes, and to restoration of consistency in cached copies of SDs. As the interface-failure rate increases beyond 30%, the rediscovery machinery tends to dominate the responsiveness results (see 5.4 for further discussion of recovery mechanisms).

5.2.2 Effectiveness

Results in Figs. 4(c) and 4(d) and the second column of Table 5 show that certain combinations lead to better update effectiveness, and Table 6 suggests that these differences could be significant. Differences in effectiveness may be partly attributed to architecture and topology. For example, 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 interface failures (two links must be operational). This suggests that a two-party architecture will be more effective under severe interface 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 the dual SCM should provide 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. Adding SCMs will likely improve the effectiveness of the three-party architecture by increasing path redundancy in the topology.

Differences in effectiveness may also be attributed in part to consistency-maintenance mechanism. In general, polling should lead to better effectiveness than notification. Our results support this for the two-party architecture and for the three-party architecture with a single SCM. Polling has built-in robustness from issuing periodic requests. On the contrary, in both two- and three-party architectures, each notification is issued only once with no further action by the sender in response to a REX (recall Table 2). In two-party notification, effectiveness suffers from situations where the notice is lost but the SM is not lost (because announcements occur only every 1800 s and thus an interface failure can be restored before the next announcement). In these situations, rediscovery does not occur and the change will not be propagated (see 5.3).

5.2.3 *Efficiency*

For a given combination of architecture and topology, we expect that notification would be more efficient than polling. We also expect that the two-party architecture would be more efficient than the three-party architecture, and that the single-SCM topology would be more efficient than the dual-SCM topology. In general, our results support these expectations, but with a few twists. The three-party, single-SCM architecture with notification proves more efficient than the two-party architectures because in Jini the SD arrives with the notification, while in UPnP notifications indicate only that a change has occurred, requiring a SU to exchange a request-response message pair to obtain the updated SD.

In notification, efficiency also decreases as the failure rate increases because SUs need to recover from REXs associated with refreshing remote registrations. Each SU must periodically refresh notification requests deposited on the SM (two-party case) or SCM (three-party case). Interface failures lead to REXs during refresh attempts. A REX invokes retry procedures: every 120 s until 540 s of continuous REX (three-party case) or every 120 s until a SM is purged (two-party case).

5.3 Investigating Saw-Tooth Phenomena

A number of the curves shown in Figures 4(a)-(f), exhibit saw-tooth phenomena, most pronounced for update effectiveness, particularly for the two-party architecture with notification. Our uncertainty calculations suggest that at failure rates above 40% these spikes may be attributed to random variations, which might be reduced by increasing the number of runs at each failure rate (currently 30) and the corresponding number of data points (currently 5 SUs x 30 runs = 150). On the other hand, spikes at lower failure rates appear more likely due to causal behavior in our models. For example, the two-party architecture with notification exhibits a significant dip at 15% interface-failure rate.

Using visualization and analysis tools included with Rapide, we examined the partially ordered sets of events (POSETs) that display the complete causal behavior of our model. The POSETs revealed that at the 15% interface-failure rate a large number of notifications were lost when either the SM transmitter was inoperable (causing notifications to all SUs to be lost) or when SU receivers were inoperable (causing lost notifications to individual SUs). Recovery from notification loss depends upon a SU discarding a SM, and then rediscovering the SM, and retrieving related SDs. A SU discards a SM when it fails to receive an announcement from the SM within the specified time. Unfortunately, in many cases, a failed interface that caused a notification loss was repaired prior to the next SM announcement (announcements come every 1800 s). In such cases, the SU does not purge the SM, and therefore there is no rediscovery. Without rediscovery, there is no mechanism to restore consistency; thus, lost notifications lead to inconsistencies that persist to the deadline (and beyond).

Why does this behavior not appear with notification in the three-party architecture? The three-party architecture requires a SM to first propagate a change to a SCM. The SCM then propagates the change on to SUs that requested notification. While notification from SCM to SU is unprotected, on failure a SM retries change propagation to a SCM. An inoperable SCM transmitter leads not only to failure to propagate notifications to SUs, but also to failure to confirm the change propagated by the SM. Absent confirmation, the SM retries the change for up to 540 s, during which time the SCM transmitter might be restored. Each repeated change that propagates to the SCM also causes notifications to be sent to the relevant SUs. Thus for SCM transmitter failures, we conclude that robustness in change propagation from SM to SCM compensates for lack of robustness in notifications from SCM to SU. No equivalent serendipity occurs in the two-party architecture. These cases suggest relationships between the timing and scope of failures and the role of recovery mechanisms in the different architectures.

5.4 Role of Recovery Mechanisms

Under hostile conditions, such as those in our experiments, recovery mechanisms play a key role in consistency maintenance. For example, a detailed analysis of results from our two-party architectural model show that at 30% failure rate and below, interface failures tend to be restored more frequently within the REX retry period associated with *HTTP Get* requests; thus, application recovery contributes substantially to update effectiveness. Above 30% failure rate, application recovery tends to exhaust its allotted time, leading a SU to discard knowledge of the SM. In such cases, update effectiveness depends primarily on robustness mechanisms built into the discovery protocol. We plan additional analysis to establish the contribution to update effectiveness of various recovery strategies in both two- and three-party architectures.

5.5 Recommendations for Improving ADLs

While the Rapide ADL provided useful abstractions to represent and analyze the structure and behavior of service-discovery protocols under failure, we recommend some improvements that apply generally to ADLs. First, this study reinforces our previous recommendations [4] that component states should be selectively exportable to allow data extraction and recording for analysis. Such an export mechanism would also assist in implementing techniques to evaluate consistency conditions that involve state variables from two or more components *and* that consider time, two important considerations when analyzing component interactions. We note that some ADLs include constraint-analysis engines that consider time [e.g., 15]. Second, ADLs, and especially their tools, must provide representations of behavior that can be evaluated efficiently. For example, to bound POSET size in this study, we were forced to substitute procedure calls in place of Rapide constraint evaluation. Third, we would find it convenient if ADL tools supported the same statistical techniques available from commercial simulation systems. For example, ADL tools might include mechanisms to track and summarize statistics about selected state variables. ADLs might also include machinery to apply statistical tests to selected variables across experiment runs in order to automate halting decisions. We expect to develop additional recommendations as our work proceeds.

6. FUTURE WORK

We envision future work along three general directions. First, we intend to complete our characterization of performance for various combinations of architecture, topology, and behavior during failures. We will model the effects of message loss, which appear likely to differ significantly from those described in this study, and we will assess the ramification of node failure on discovery and recovery mechanisms in various architectures and topologies. Second, we plan to model and evaluate selected changes to improve the performance of discovery architectures and protocols in response to failure. Here, our goal is to increase the fault-tolerance of such systems. We intend to implement and evaluate our most promising suggested changes in publicly available service-discovery software. Third, we will expand our generic structural model of service-discovery architectures to include message exchanges and verifiable consistency conditions.

Along a different dimension, we hope to improve methodologies available to design and engineer distributed software systems. At present, many publicly available specifications come with one or more reference implementations. We hope to demonstrate that architectural models lead to better understanding of the properties of distributed systems. In addition, we aim to improve ADLs, and associated tools, by providing recommendations based on our experience. We are also considering developing our own modeling and analysis tools especially designed for understanding collective behavior in multi-party distributed systems.

7. CONCLUSIONS

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. While likely suitable for small-scale commercial applications, questions remain regarding the performance of such protocols at large scale, and during periods of high volatility and duress, such as might exist in military and emergency-response applications. In this paper, we used architectural models to characterize the performance of selected combinations of system topology and consistency-maintenance mechanism during catastrophic communication failure. Further, we used behavioral analysis to investigate causes underlying observed performance. Our initial investigations show significant differences in update effectiveness can be obtained by varying aspects of the design (architecture, topology, consistency-maintenance mechanism, and recovery strategies). Our results also suggest relationships among interface-failure rate, failure timing, and recovery strategies.

8. ACKNOWLEDGMENTS

The work described benefits from financial support provided by the National Institute of Standards and Technology (NIST), the Defense Advanced Research Projects Agency (DARPA), and the Advanced Research Development Agency (ARDA). In particular, we acknowledge the support of Susan Zevin from NIST, Doug Maughan and John Salasin from DARPA, and Greg Puffenbarger from ARDA. We also thank Stefan Leigh of NIST and the anonymous WOSP reviewers for insightful comments that helped us to improve the manuscript.

9. REFERENCES

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

[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] 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.

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

[6] Salutation Architecture Specification, Version 2.0c, Salutation Consortium, June 1, 1999.

[7] Specification of the Home Audio/Video Interoperability (HAVi) Archiecture, V1.1, HAVi, Inc., May 15, 2001.

[8] Service Location Protocol Version 2, Internet Engineering Task Force (IETF), RFC 2608, June 1999.

[9] Specification of the Bluetooth System, Core, Volume 1, Version 1.1, the Bluetooth SIG, Inc., February 22, 2001.

[10] B. Miller and R. Pascoe, Mapping Salutation Architecture APIs to Bluetooth Service Discovery Layer, Version 1.0, Bluetooth SIG White paper, July 1, 1999.

[11] C. Bettstetter and C. Renner, "A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol", *Proceedings of the Sixth EUNICE Open European Summer School: Innovative Internet Applications*, EUNICE 2000, Twente, Netherlands, September, 13-15, 2000.

[12] G. Richard, "Service Advertisement and Discovery: Enabling Universal Device Cooperation," *IEEE Internet Computing*, September-October 2000, pp. 18-26.

[13] B. Pascoe, "Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun: How does the Salutation Architecture stack up," Salutation Consortium whitepaper, June 6, 1999.

[14] J. Rekesh, UPnP, Jini and Salutation - A look at some popular coordination framework for future network devices, Technical Report, California Software Lab, 1999. Available online from http://www.cswl.com/whiteppr/tech/upnp.html.

[15] Allen, R. A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997.