Understanding Consistency Maintenance in Service Discovery Architectures in Response to Message Loss

Christopher Dabrowski, Kevin Mills, Jesse Elder
National Institute of Standards and Technology
Gaithersburg, Maryland USA
{cdabrowski, kmills, 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 severe message loss. 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 message-loss 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?).

1. Introduction

Successful deployment of active middleware services, which can detect and adapt to changes in topologies of distributed components, will depend upon a foundation layer of service-discovery software that can monitor the state of nearby software services and components and that can detect changes in network connectivity. Already, military organizations are investigating the applicability of commercial service-discovery systems to meet such requirements in hostile and volatile environments [1]. In military and civil emergency response situations, software components in a distributed system may find that cooperating components disappear due to physical or cyber attacks, to jamming of communication channels or to movement of nodes. Such environments demand new analysis approaches and tools to design and test software that will be used to provide active middleware services.

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 severe message loss. (A companion paper investigates robustness in the face of interference due to node interface failure [2].) 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] [3] and Universal Plug-and-Play (UPnP) [4]. We use our models to investigate performance differences among combinations of architecture and consistency-maintenance mechanism as message-loss 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 [5] where we derived benefits by creating dynamic models from specifications for service-discovery protocols. Dynamic models 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

[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.

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 [6], 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.

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. 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 message loss. In Section 5, we present results from the experiment, and we discuss causes underlying some of the results. We conclude in Section 6.

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 [3, 4, 7-10]. 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 [3], a general service-discovery mechanism atop Java™, which provides a base of portable software technology. As another example, a group of engineers at Microsoft and Intel conceived Universal Plug-and-Play [4] 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). 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 graphical-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) [9]. The behaviors by which SUs discover and maintain consistency in desired SDs depend partly upon the service-discovery architecture employed.

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)

Table 1. Mapping concepts among service-discovery systems.

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 or 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 using the lazy discovery process), 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.

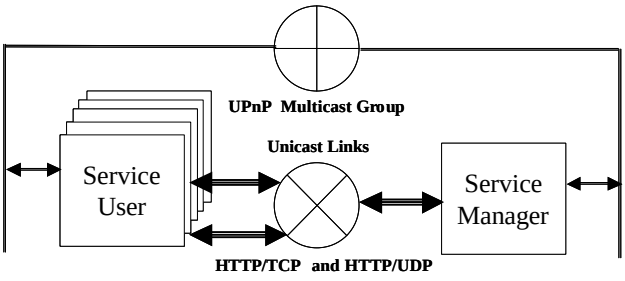


Figure 1. Two-party service-discovery architecture deployed in a six-node topology: five service users (SUs) and one service manager (SM).

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 chose behaviors described in the Jini specification.

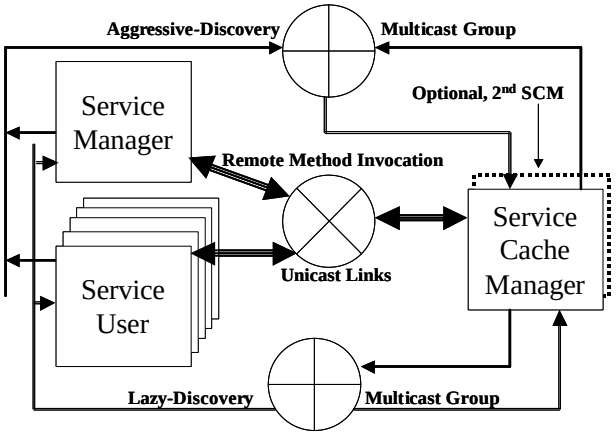


Figure 2. Three-party service-discovery architecture deployed in a seven- or eight-node topology: five service users (SUs), a service manager (SM), and a service cache manager (SCM), with an optional 2nd SCM.

In Jini, the discovery process focuses upon discovery by SMs and SUs of any intermediary SCMs that exist in the network neighborhood. Elsewhere [5], 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. 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 descriptions of 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 updates.

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-s 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.

3. Modeling recovery strategies

Elsewhere [2], we discuss the classes of network failures occurring in hostile environments and describe failure-recovery mechanisms of lower-layer protocols in more detail. Here we address recovery in response to message loss at a more general level. 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.

3.1 Recovery by lower layers

Our models operate over two types of channels: unreliable, simulating the UDP in both multicast and unicast forms, and reliable, simulating the TCP. UDP provides no guarantee of message delivery; therefore our simulated unreliable channels discard messages lost due to transmission errors. Neither sender nor receiver learns the fate of lost messages.

Responsible Party	Recovery Mechanism	Two-Party Architecture (UPnP)	Three-Party Architecture (Jini)
Lower-Layer Protocols	UDP	No recovery	No recovery
	TCP	Issue REX in 78 s if connection establishment fails	Issue REX 78 s if connection establishment fails
Discovery Protocols	Lazy Discovery	SM: announces with n ($3+2d+k$) 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

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

Reliable unicast protocols attempt to ensure delivery of messages by detecting and retransmitting lost messages. Accordingly in the TCP simulation, our model is more complex, including both connection establishment and data transfer. During connection establishment, we allow up to four attempts to initiate a connection. An attempt fails if either the connection request or accept is lost. If no accept arrives, then the request is resent in 6 s for the first retry, but we wait 24 s for each subsequent retry. If all attempts fail, then we signal a REX to the requester. During data transfer, messages lost to transmission errors are scheduled for retransmission (roughly within a round-trip time, or RTT). We increase the retransmission timeout by 25% with each successive retransmission. We place no bound on the number of retransmissions during data transfer.

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 lazy announcement recurs periodically. Receiving nodes can cache information from the announcements; the cached information may be purged if communication fails. 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 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 if 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. All retries occur until some time bounds, after which knowledge of the discovery is discarded.

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 message loss? 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 compare change propagation in two- and three-party architectures, we then combine the architectures with different consistency-maintenance mechanisms. Table 3 depicts the six combinations. 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 message-loss rate up to 95% (in increments of 5%). We provide further details below.

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)

Table 3. Experiment combinations.

4.1. Tracking consistency

To track consistency in our experiment, we employ property analysis [5], 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, expressed as:

FOR All (SM, SU, SD)
(SM, SD [Attributes1]) isElementOf SM managed-services &
(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.

4.2. Generating message loss

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 define F as the message-lost rate, which represents the independent variable in our experiment, ranging from 0.00 to 0.95 in increments of 0.05. For each attempt to transmit a data message, whether on a reliable or unreliable channel, or to retransmit a data message on a reliable channel, or to send or retry a connection request or accept message on a reliable channel, we select a uniform random number, V, from the unit interval 0 to 1. If $V < F$, we discard the message, which in the case of messages sent on the reliable channel will stimulate a retransmission after the appropriate timeout period (recall 3.1). Table 4 summarizes most of the relevant parameters and values for our experiments.

	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
Message loss parameters and protocol response	Loss Probability (*F*)	Each transmission attempt fails with *P*(*F*)
	Unreliable protocol response	Message discarded. No retransmission.
	Reliable protocol response	Connection Establishment - 4 retransmission attempts with delays of 6 s, 24 s, 24 s and 24 s; then REX if unsuccessful. Data Transfer – retransmit until success, increasing time-out by 25% on each retry (first time-out is round-trip time).
	Transmission delay without message loss	0.14 – 0.42 s uniform (range is multiplied by 1+*F*)
	Per-item processing delay within node	100 us for cache items 10 us for other items

Table 4. Values for relevant parameters.

4.3. Metrics

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

4.3.1 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*.

4.3.2 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 4.3.1 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: $O = X * N$. 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 $j = 1...N$.

4.3.3 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 topologies, *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 and propose reasons for these differences.

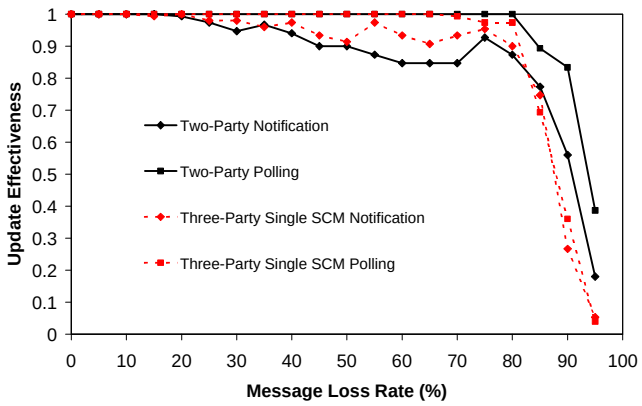
5.1. Results

In a series of six graphs, which have identical abscissas (message-loss rate, increasing from 0% to 95% in increments of 5%) and ordinates (an appropriate metric 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 (average), effectiveness, or efficiency (average). Figure 3(a) compares effectiveness from our two-party model against that from our single-SCM, three-party model, for both polling and notification. Figure 3(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 3(c) and 3(d) compare update responsiveness using the same combinations. Figures 3(e) and 3(f) use the same combinations, but compare update efficiency. The graphs reporting measures of effectiveness and responsiveness 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 message losses) and then asymptotically approaches zero efficiency as the message-loss rate increases toward 100%. 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 message-loss rates, for a specified configuration.

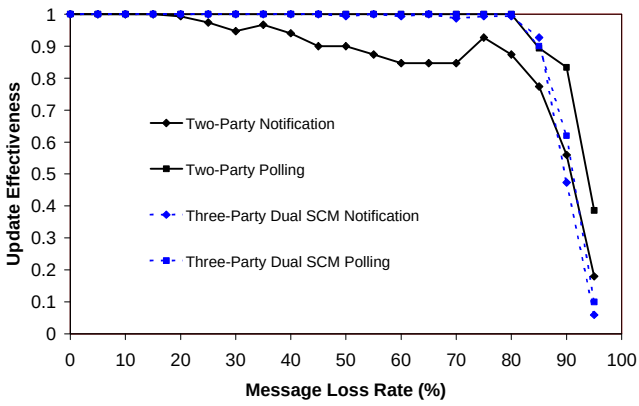
5.2. Relative performance

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

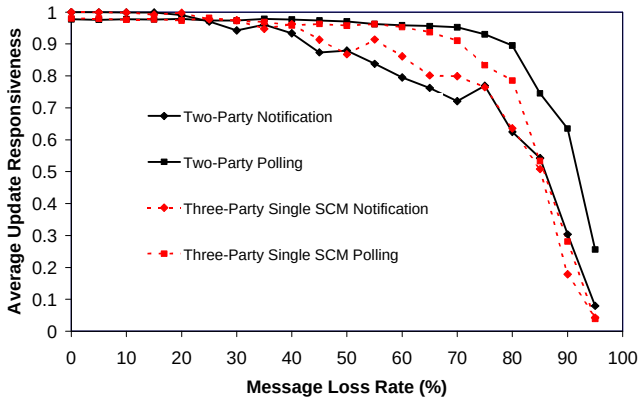
5.2.1 Effectiveness. Figs. 3(a) and 3(b) show that all combinations of architecture, topology, and consistency maintenance strategy exhibit update effectiveness of 0.85 or better up to a message-loss rate of 85%, after which they decline sharply. This similarity in effectiveness among the combinations can be attributed to commonality in the recovery behaviors of the discovery protocols, as implemented in our models. We require each SU (and the SM in the three-party case) to discard discovered information after a break in communications (recall Table 2) and then to initiate rediscovery. In the two-party model, periodic (120 s) *Msearch* queries by each SU (aggressive-discovery procedures) lead to rediscovery. Similarly, in the three-party case, periodic (120 s) announcements by each SCM (lazy-discovery procedures) lead to rediscovery.



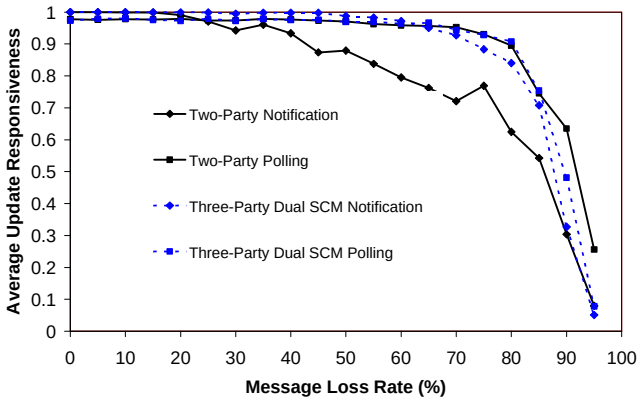
(a) Update effectiveness of two-party vs. three-party (single-SCM)



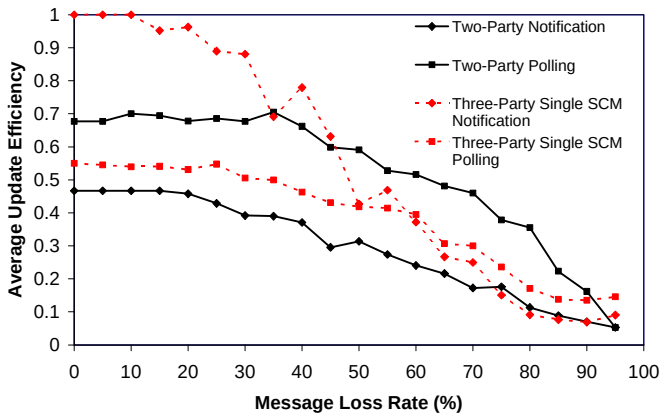
(b) Update effectiveness of two-party vs. three-party (dual-SCM)



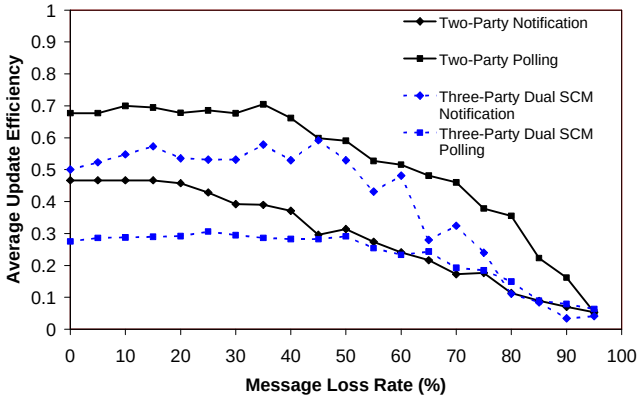
(c) Average update responsiveness of two-party vs. three-party (single-SCM)



(d) Average update responsiveness of two-party vs. three-party (dual-SCM)



(e) Average update efficiency of two-party vs. three-party (single-SCM)



(f) Average update efficiency of two-party vs. three-party (dual-SCM)

Figure 3. Graphs comparing combinations of architecture, topology, and consistency-maintenance mechanism.

After rediscovering a discarded node, the SU or SM re-establishes lost registrations, as appropriate for the consistency-maintenance strategy: notification registration for SUs and service registrations for the SM (three-party cases). In the process of restoring this distributed state information, each SU may obtain and cache a consistent copy of the SD maintained by the SM. As message-loss rate increases beyond 50%, this rediscovery machinery tends to dominate the effectiveness results.

	Mean (across all message-loss rates)		
	Update Effectiveness	Average Update Responsiveness	Average Update Efficiency
Two-Party Notification	0.867	0.799	0.296
Two-Party Polling	0.956	0.901	0.525
Three-Party Notification (Single SCM)	0.870	0.807	0.552
Three-Party Polling (Single SCM)	0.902	0.846	0.391
Three-Party Notification (Dual SCM)	0.921	0.881	0.400
Three-Party Polling (Dual SCM)	0.931	0.887	0.233

Table 5. Summary statistics (mean across all message-loss rates) computed for each curve in the graphs shown in Figures 3(a) through 3(f).

Despite rough similarity, certain combinations do show slightly better effectiveness than others (see Figs. 3(a) and 3(b) and the first column of Table 5). We attribute these differences to the consistency-maintenance strategy (polling or notification), and to differences in the recovery actions taken by the application software while implementing a particular strategy. Architecture and topology play a secondary role. In general, polling should lead to better effectiveness than notification, and our results support this in all architecture-topology combinations. Polling has built-in robustness from issuing periodic requests. On the contrary, notifications are issued only once with no further action by the sender in response to a REX (recall Table 2). Therefore, in notification, effectiveness suffers from situations where the notice is lost but where the notification registration and the node (SM or SCM) discovery are not lost. In these situations, there is no opportunity for recovery mechanisms to regain a lost node (SM or SCM) and to register for notification. Without such recovery, the SU might never obtain a copy of a changed SD. However, in three-party notification with dual SCMs, the effects of architecture and topology also come into play. Here, a replicated SCM provides an additional path for the SM to propagate the update, thus increasing the effectiveness of notification almost to the level of polling.

Beyond a rough similarity with distinguishable differences, the curves for effectiveness in two-party notification and in three-party single-SCM notification also include some irregularities, where effectiveness first drops and then improves as the message-loss rate increases. We used Rapide analysis tools to investigate the reasons underlying these dips. For both cases, we found that as the failure rate increases beyond 40%, the rate of recovery of the lost SM and lost registrations also increases. Recall that notification has no built-in robustness, relying instead on recovery mechanisms in TCP. Thus, to regain consistency when TCP recovery fails, notification must rely on recovery mechanisms in the discovery protocols, which provide opportunities to propagate previously lost updates. The higher the recovery rates, the greater the number of opportunities to regain consistency. As the message-loss rate increases, the recovery rate increases, and the effectiveness improves, up to a limit. Once the message-loss rate reaches 80%, the ability of the discovery protocols to effect recovery becomes impaired, leading to an inevitable decline in effectiveness. We also note that between 40% and 80% message-loss rate one of the notification combinations (three-party single-SCM) provides better effectiveness than the other (two-party). We suspect this occurs because the recovery actions of the SM (regaining the SCM discovery and registering the SD) provide additional opportunities (not available in the two-party case) to propagate the updated SD. Also recall that in Jini (the basis for behavior in our three-party models) notification includes the SD, while in the two-party case, based on UPnP, the SU must invoke separate operations to retrieve a copy of the SD. This provides additional opportunities for message loss to interfere with the restoration of consistency in the two-party case. These somewhat surprising dips in the effectiveness curves for notification also appear under conditions of node interface failures, discussed in a companion paper [2].

5.2.2 Responsiveness. Results in Figs. 3(c) and 3(d) and the second column of Table 5, show that three combinations of architecture and behavior (two-party polling, three-party polling with dual SCMs, and three-party notification with dual SCMs) exhibit similar responsiveness. Below 70% message-loss rate, three-party polling with a single SCM also exhibits similar responsiveness, but then declines more steeply than the others. For each architecture-topology combination, Table 5 shows that polling leads to better overall responsiveness than notification. However, Figs. 3(c) and 3(d) show that notification is more responsive at lower message-loss rates, where the periodicity of polling incurs a greater lag time. As message-loss rate increases, polling becomes more responsive than notification, which must rely on recovery mechanisms in the discovery protocols to recover from failure to transfer notifications (recall 5.2.1), whereas the built-in robustness of polling overcomes failures in lower protocol layers. In the three-party case with dual SCMs, notification achieves a similar

responsiveness to polling because notifications are sent over redundant paths, which mitigate the effect of transmission failures.

At high message-loss rates, under both polling and notification, restoring consistency depends largely upon recovery mechanisms in the discovery protocol. For responsiveness, as for effectiveness, our models of these recovery mechanisms ensure a degree of similarity in the results for three cases: two-party polling, three-party polling, and three-party notification with dual SCMs. In the case of three-party polling with a single SCM, responsiveness declines more rapidly at higher message-loss rates because, lacking a redundant SCM, fewer opportunities exist to recover a copy of the updated SD. Finally, for reasons already addressed (see 5.2.1), between 40% and 90% message-loss rates, both two-party notification and three-party notification with a single SCM prove considerably less responsive than the other combinations.

5.2.3 Efficiency. For a given combination of architecture and topology, we expect notification to be more efficient than polling. We also expect the two-party architecture to be more efficient than the three-party architecture, and the single-SCM topology to be more efficient than the dual-SCM topology. In general, our results support these expectations. However, there are a few twists. First, 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 the notifications indicate only that a change has occurred, requiring a SU to exchange a request-response message pair to obtain the updated SD. Second, each SU must periodically refresh notification requests deposited on the SM (two-party case) or SCM (three-party case). As the message-loss rate increases, failure to transfer refresh messages leads to REXs, which stimulate 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). For this reason, efficiency decreases for notification as the message-loss rate increases.

6. 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 node connectivity. 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 severe message loss. Further, we used behavioral analysis to investigate the causes of observed performance. Our initial investigations show significant differences in performance can be obtained by varying aspects of the design (architecture, topology, consistency-maintenance mechanism, and recovery strategies).

7. 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 and Scott Rose of NIST and the anonymous WAMS reviewers for insightful comments that helped us to improve the manuscript.

8. 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] Dabrowski, C., Mills, K., and Elder, J. "Understanding Consistency Maintenance in Service Discovery Architectures during Communication Failure", *Proceedings of the 3rd International Workshop on Software Performance*, ACM, Rome, Italy, July 24-26, 2002.

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

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

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

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

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

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

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