Available online at www.sciencedirect.com
ScienceDirect

The Journal of Systems and Software xxx (2007) xxx–xxx

The Journal of Systems and Software
www.elsevier.com/locate/jss

Understanding failure response in service discovery systems

C. Dabrowski *, K. Mills, S. Quirolgico

National Institute of Standards and Technology, Information Technology Laboratory, 100 Bureau Drive, Mailstop 8970, Gaithersburg, MD 20899-8970, United States

Received 1 March 2006; received in revised form 21 November 2006; accepted 23 November 2006

Abstract

Service discovery systems enable distributed components to find each other without prior arrangement, to express capabilities and needs, to aggregate into useful compositions, and to detect and adapt to changes. First-generation discovery systems can be categorized based on one of three underlying architectures and on choice of behaviors for discovery, monitoring, and recovery. This paper reports a series of investigations into the robustness of designs that underlie selected service discovery systems. The paper presents a set of experimental methods for analysis of robustness in discovery systems under increasing failure intensity. These methods yield quantitative measures for effectiveness, responsiveness, and efficiency. Using these methods, we characterize robustness of alternate service discovery architectures and discuss benefits and costs of various system configurations. Overall, we find that first-generation service discovery systems can be robust under difficult failure environments. This work contributes to better understanding of failure behavior in existing discovery systems, allowing potential users to configure deployments to obtain the best achievable robustness at the least available cost. The work also contributes to design improvements for next-generation service discovery systems.
Published by Elsevier Inc.

Keywords: Distributed systems; Robustness; Service discovery

1. Introduction

Various teams designed and implemented a first generation of (competing) service discovery systems that enable distributed components to find each other without prior arrangement, to express capabilities and needs, to compose into collections, and to detect and adapt to changes. Each specific design defines a system structure, along with protocols for discovery, monitoring, and recovery. Some designs assume a specific underlying communication technology, some designs focus on one application domain, and some designs were conceived to operate over Internet protocols and to support many applications.

In this paper, we investigate the architectures and behaviors underlying Jini Networking Technology[1] (Arnold, 1999), Universal Plug and Play (UPnP) (Universal Plug and Play Device Architecture, 2000), and the Service Location Protocol (SLP) (Guttman et al., 1999) when subjected to various failures. Elsewhere (Dabrowski et al., 2005), we present a generic model encompassing the designs of these systems and we identify performance issues that could arise. While this previous work considers system behavior absent failures, here we explore the relative ability of discovery systems to cope with different types and intensities of failure.

We reported preliminary results in various conference papers (Dabrowski and Mills, 2001; Dabrowski et al.,

[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.

* Corresponding author. Tel.: +1 301 975 3249; fax: +1 301 948 6213.
E-mail address: cdabrowski@nist.gov (C. Dabrowski).

0164-1212/$ - see front matter Published by Elsevier Inc.
doi:10.1016/j.jss.2006.11.017

2002a,b, 2003); however, this paper improves upon earlier work in two ways. First, we extend the scope of our results to cover three architectures (two-party, three-party, and adaptive), three failure scenarios (configuration restoration, service acquisition and maintenance, and consistency maintenance), four failure types (power failure and restart, node failure, communication failure, and message loss), and a set of failure detection and recovery techniques at three levels (transport protocols, discovery protocols, and application logic). Second, we increase the amount of data collected and analyzed to obtain better estimates for performance metrics at high failure rates.

This paper contributes to the understanding of service discovery systems. First, this paper characterizes robustness of discovery systems under difficult failure environments. This paper further identifies and discusses the most significant design and configuration decisions that influence robustness. Second, this paper identifies specific design and deployment decisions that could lead to diminished robustness. Third, this paper quantifies the relative cost associated with specific decisions. Overall, the information provided here should contribute to better understanding of failure behavior in existing discovery systems, allowing potential users to configure deployments to obtain the best achievable robustness at the least available cost. Further, results and discussions presented here have contributed to design improvements in the next generation of discovery systems (Sundramoorthy, 2006; Sundramoorthy et al., 2004).

This paper also contributes experimental methods to study robustness in distributed systems. First, we introduce and apply metrics to quantify relative robustness and cost at the application level for various scenarios. Second, we present a technique to decompose aggregate robustness into detection and recovery latency. Using this technique, we show how similar robustness can be achieved through different behaviors arising from particular design choices. Our methods can be adopted, adapted, or extended by other researchers to investigate failure response in distributed systems – a topic due for increased study.

We begin (in Section 2) with a synopsis of existing work comparing and contrasting service discovery systems. Most previous work focuses on functional comparisons, on means for translating among discovery systems, or on improving existing designs. Our own related work (Dabrowski et al., 2005; Bowers et al., 2003; Mills and Dabrowski, 2003; Rose et al., 2003; Mills et al., 2004; Tan and Mills, 2005) attempts to unify designs for several existing discovery systems, and investigates performance problems arising when such systems are deployed at large scale.

In Section 3, we survey the design and function of service discovery systems. We introduce a model to convey concepts across selected systems. Using our model, we describe how discovery operates under UPnP (a two-party architecture, where clients issue multicast queries to find services), Jini (a three-party architecture, where clients consult a directory to find services), and SLP (which is a three-party architecture that can adapt to become a two-party architecture). We also describe two mechanisms (polling and notification) used by discovery systems to maintain consistent information among distributed replicas. The architectures, discovery procedures, and consistency maintenance mechanisms described in Section 3 form the basis for scenarios, experiments, and results recounted in later sections.

In Section 4, we introduce selected types of failure that can impede a distributed system and we discuss selected techniques to detect and recover at three layers. At the lowest layer, transport protocols may include detection and recovery mechanisms (e.g., acknowledgments, retransmissions, and exceptions). In the middle layer, discovery protocols typically include some detection and recovery mechanisms (e.g., heartbeats and soft state). At the top layer, applications may take recovery actions in reaction to exceptions raised by transport protocols. Interactions among these detection and recovery techniques can become quite intricate and difficult to understand.

In Section 5, we describe our experiment methodology, consisting of six steps: (1) constructing (simulation) models reflecting structure, behavior, and deployments of selected service discovery systems, (2) incorporating failure models into the simulations (3) devising scenarios and related metrics to quantify robustness and cost, (4) simulating scenarios for selected configurations over a range of failure rates, (5) collecting, analyzing, and plotting data from simulations, and (6) investigating unexpected results and anomalies. In Section 6, we describe the design and results for our experiments: (1) restart after power failure, (2) service acquisition and maintenance impeded by node failures, and consistency maintenance impeded by (3) communication failures and (4) message loss. We report results from these four experiments, which encompass 30 configurations. For each experiment, we explain the scenario and failure model, define metrics, present results, outline findings, and discuss unexpected outcomes. We close in Section 7 with a précis of our findings and contributions.

2. Related work

Emergence of various specifications for service discovery systems, coupled with the anticipated importance of discovery functionality in future distributed systems, has stimulated significant interest in understanding similarities and differences among competing designs. Most existing comparisons focus on architecture, features, and function. A few comparisons also consider programming differences, because most discovery systems are conceived as middleware to support distributed applications. Bettstetter and Renner (2000) compare SLP, Jini, UPnP, and Bluetooth with respect to architecture, function, and features, and consider underlying requirements for programming languages, operating systems, and network protocols. The comparison is expressed using concepts and terminology specific to each discovery system, although the authors do identify three common aspects (support for searching on service attributes, inclusion of a directory, and use of leasing) for comparison. Richard (2000) compares software architectures, along with system features and functions, for Jini, Bluetooth, Salutation, SLP, and UPnP. Elsewhere, Richard (2002), expands his comparison to include programming considerations by providing source code for clients and services in Jini, SLP, UPnP, and Bluetooth. Pascoe (1999) outlines a brief architectural comparison of Jini, UPnP, and Salutation, and Rekesh (1999) gives a similar comparison that appears to be based on Pascoe's work. In a subsequent paper, Pascoe (2001) amplifies his architectural comparison to include comparison of functions and features. O'Driscoll (2000), when considering a wide range of home networking technology, provides descriptions of Bluetooth, HAVi (the Home Audio-Video interoperability specification), UPnP, and Jini. Though giving no direct comparison, O'Driscoll provides a summary of architecture, function, and features from which readers may infer a comparison. Olivier (2000) provides a detailed description of Jini, but also includes a brief description of UPnP and a comparison between Jini and SLP. None of these comparisons considers performance or robustness.

Limitations in existing comparisons motivated our own work. Elsewhere (Dabrowski et al., 2005), we provide a unified and general model for first-generation discovery systems and then show how our model can be used to represent Jini, UPnP, and SLP. Our unified model, conceived with neutral terminology, provides a basis for direct comparison among architectural, functional, and behavioral elements of designs. Our model also reveals limitations and open issues in existing designs and specifications, and includes a set of service guarantees that we believe discovery systems should attempt to satisfy. Further, we identify selected performance issues that may arise when deploying discovery systems at large scale, and we use our model to outline algorithms that might improve performance. While our previous work improves on existing comparisons, we did not consider robustness under various types of failure. The present paper extends our previous work by comparing failure response in the major designs for first-generation discovery systems (as represented by Jini, UPnP, and SLP).

As a natural extension to functional comparisons, some researchers conceive protocol translators in order to achieve interoperation among dissimilar service discovery systems. For example, the Open Services Gateway Initiative (OSGi) [see Bushmitch et al., 2004, and also chapter 17 in O'Driscoll, 2000] defines a layer of middleware to bridge among Jini, UPnP, and Bluetooth. Miller and Pascoe (1999) show how to map between the application-level programming interfaces of Salutation and Bluetooth. Allard et al. (2003) and Sameh and El-Kharboutly (2004) describe different techniques to bridge between Jini and UPnP, while Guttman and Kempf (1999) consider techniques to bridge between Jini and SLP. Similarly, Yu et al. (2003) define a software structure for middleware that can bridge among a diverse set of service discovery systems and distributed object systems. Ponnekanti and Fox (2003) take a more general tact by defining a framework that clients may use to find candidate services and to automatically configure an appropriate set of proxies and stubs to allow a client to invoke a selected service. Only one (Sameh and El-Kharboutly, 2004) of these papers investigates performance, and none considers the effects of failures. While our paper does not consider translation among discovery systems, researchers could use our method to investigate and quantify robustness of various designs for bridges and translators.

Beyond first-generation systems for discovery of services operating in close proximity, researchers in industry and academe are investigating how to build discovery systems that scale over a wide area. An early proposal, known as Universal Description, Discovery and Integration (UDDI, 2000), defines well-known, web-accessible repositories, where service descriptions may be deposited so that clients may query for services of interest. The UDDI approach exhibits limited scalability because every service in a network must deposit its description with a central directory, or else with multiple replicas of a central directory. To overcome such limitations, researchers continue to propose a number of more flexible approaches. One early idea, E-speak (Frolund, 2000), used an expanding-ring multicast search to discover directories that are organized into a federated topology through which service descriptions permeate over time. A similar idea is contained in JXTA (2004), where a peer-to-peer system is used to disseminate copies of service descriptions throughout a topology of caches, and in Neuron (Hsiao and King, 2002), a self-organizing and self-tuning topology of caches that can tolerate failures of nodes and communication links. Other self-organizing directories have also been proposed, including SRIRAM (Verma, 2003), NeuroGrid (Joseph, 2002), and the Secure Service Discovery Service (Czerwinski, 1999). A somewhat different approach (Castro, 2002) forms a logical ring (based on node addresses) that helps individual nodes to bootstrap into various available overlay networks, each of which advertises services. Grid researchers have also proposed a design for wide-area service discovery (Iamnitchi and Foster, 2001), coupled with the ability to inject and disseminate real-time status information (Czajkowski et al., 2001). Most of these designs include provisions to detect and recover from failures or to mitigate failures; however, no comprehensive results exist that compare robustness among various designs. While this paper investigates robustness only for local discovery, we suspect that our method could be applied to quantify and compare robustness among designs for wide-area discovery.

3. Modeling service discovery systems

Service discovery systems enable components in a network to discover each other, and to determine if discovered components meet specific requirements. Further, discovery systems 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 distributed components. Many diverse industry activities explore different approaches to meet such requirements, leading to a variety of proposed designs (Salutation Architecture, 1999; Arnold, 1999; Universal Plug and Play Device Architecture, 2000; Guttman et al., 1999; Home Audio/Video Interoperability (HAVi) Architecture, 2001; Bluetooth System, 2001). Some groups approach the problem from a vertically integrated perspective, coupled with a narrow application focus. Other groups propose more widely applicable solutions. For example, a team of researchers and engineers at Sun Microsystems designed Jini Networking Technology (Arnold, 1999), a discovery system 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 (UPnP) (Universal Plug and Play Device Architecture, 2000) to extend plug-and-play from single computers to distributed systems. Similarly, the efforts of Sun Microsystems and other companies led to the Service Location Protocol (SLP) (Guttman et al., 1999), aimed at providing service discovery for the Internet.

While these designs appear quite different, the systems share some common traits. First, they all assume availability of the Internet protocols as a base. Second, they all provide general approaches to describe the capabilities and status of services. Third, they all include mechanisms that can be used to detect and recover from failures. Jini, UPnP, and SLP differ in architecture, in approach to describing services, and in assumptions about how to use transport protocols. This interesting combination of similarities and differences led us to base our comparative study on Jini, UPnP, and SLP. Our main challenge was finding a means to clearly understand and represent similarities and differences among the three systems. To address this challenge, we developed a general model with common terminology and then mapped concepts from each specific system into our model.

3.1. A general model of service discovery systems

Our model provides a basis for comparative analysis of various discovery systems by representing major architectural components and concepts with a consistent and neutral terminology (see first column in Table 1). The main components in our model include: (1) *service user*, (2) *service manager*, and (3) *service cache manager*. A service user (SU) is a client in a service discovery system. A SU is concerned with discovering services from components within the distributed system, acquiring access to discovered services, and using discovered services. A service manager (SM) maintains a database of *service descriptions*, each of which encodes the characteristics of a particular *service provider* (i.e., the provider of the service). Each service description (SD) contains the identity, type, and attributes that characterize a service provider (SP). Each SD also includes the addresses of software interfaces (e.g., an application-programming interface or graphic user interface) to access a service. A SU seeks SDs satisfying specific requirements. A service cache manager (SCM) operates as an intermediary, matching advertised SDs from SMs to requirements provided by SUs. SCMs are optional components supported by some, but not all, discovery systems. Table 1 shows how these general concepts map to specific concepts from Jini, UPnP, and SLP.

The behaviors by which (Jini, UPnP, and SLP) SUs discover and maintain consistency in relevant SDs depend in part upon the system architecture and design and in part on the transport protocols used. Transport protocols are used for two kinds of message exchange: (1) multicast, in which transmitted messages are conveyed to all receivers that participate in a multicast group and (2) unicast, which is point-to-point communication directly between a pair of corresponding entities. Both Jini and UPnP use the User Datagram Protocol (UDP) for exchanging multicast messages and use the Transmission Control Protocol (TCP) for exchanging unicast messages. UPnP also uses UDP to unicast answers to multicast queries. SLP uses UDP for exchanging both multicast and unicast messages. The differences in transport protocols become significant when considering approaches to detect and recover from failures;

Table 1
Mapping concepts among selected service discovery systems

Generic model	Jini	UPnP	SLP
Service user	Client	Control point	User agent
Service manager	Service or device proxy	Root device	Service agent
Service provider	Service	Device or service	Service
Service description	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	Lookup service	Not applicable	Directory service agent (optional)

therefore, we defer (until Section 4) a more detailed discussion. Here, we focus on behavioral differences arising from variations in architecture and design.

3.2. Modeling service discovery architectures and protocols

Our analysis of six distinct discovery systems revealed that most designs use one of two architectures: *two-party* or *three-party*. One discovery system we examined uses both architectures together. A two-party architecture consists of two major component types: SMs and SUs. Fig. 1 illustrates a two-party architecture (configured for UPnP). Service discovery occurs through interactions between these two component types; SUs discover SMs and then query them for suitable SDs. A three-party architecture adds a third component type, the SCM, which contains a directory. Fig. 2 illustrates a three-party architecture (configured for Jini). In a three-party architecture, both SMs and SUs first discover SCMs to serve as intermediaries. SMs deposit SDs with SCMs and SUs interact with SCMs to obtain suitable SDs. A third architectural variant (supported by SLP) employs both the two-party and three-party architecture and is capable of switching between them, depending on circumstances. We call this an *adaptive* architecture.

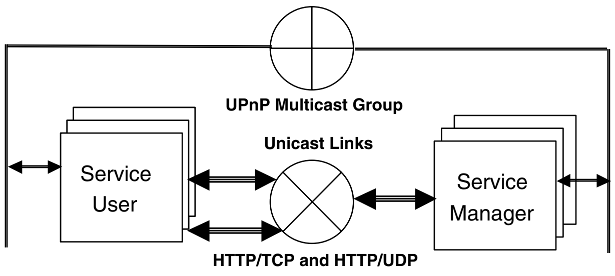


Fig. 1. Two-party service discovery system deployed in a topology with three service users (SUs) and three service managers (SMs).

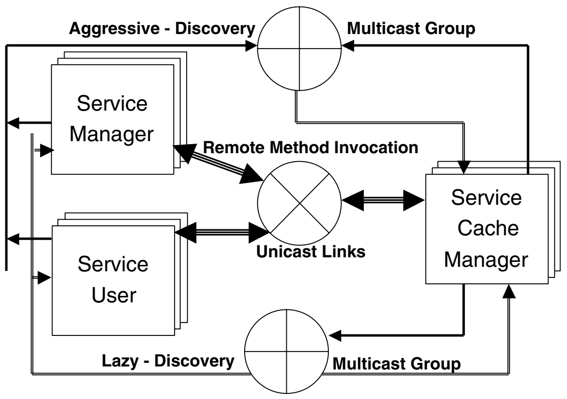


Fig. 2. Three-party service discovery system deployed in a topology with three service users (SUs), three service manager (SMs), and three service cache manager (SCMs).

3.2.1. Discovery in two-party architectures

Given a two-party architecture, we model the behavior of participating SMs and SUs. Upon startup, each SU and SM engages in a discovery process to locate other relevant components within the network neighborhood. We chose behaviors described in the specification for UPnP (Universal Plug and Play Device Architecture, 2000).

In a lazy-discovery process, each SM periodically announces existence of its SDs over a designated UPnP multicast group. Upon receiving these announcements, SUs with matching requirements use a HyperText Transfer Protocol (HTTP)/TCP unicast link to request, directly from the SM, copies of the SDs associated with relevant SPs. The request is made using an HTTP GET request. 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 unicast link to respond (after a jitter delay) directly to the SU. Whenever a SM responds to an `Msearch query` (or announces itself), it repeats a sequence of messages, with separate messages for distinct devices and service types managed by the SM. For each appropriate response, the SU uses a HTTP/TCP unicast link to send an HTTP `GET` request for a copy of relevant SDs, caching them locally.

In UPnP, multiple HTTP `GET` requests are required to transfer the SD, because each SD consists of two parts. 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 (1800 s minimum recommended). Each announcement specifies a 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.

3.2.2. Discovery in three-party architectures

Given a three-party architecture, we model the behavior of participating SCMs, SMs, and SUs, which each engage in a discovery process upon startup. We chose behaviors described in the Jini specification (Arnold, 1999), where SMs and SUs attempt to discover any intermediary SCMs that exist in the network neighborhood.

Upon initiation, a Jini component enters aggressive discovery, where it transmits probes on a designated 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 a designated 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. SUs may query discovered SCMs for SDs of interest. Alternatively, a SU may deposit a query with the SCM, which will attempt to match SDs provided by SMs to specifications of the deposited query. The SCM forwards any matching SDs on to the SU that deposited the relevant query.

3.2.3. *Discovery in adaptive architectures*

An adaptive architecture requires SMs and SUs to rendezvous through a SCM, but allows direct SM–SU interaction when no SCM is available. If SMs and SUs interact directly and a SCM becomes available, then the architecture requires SMs and SUs to resume interacting through the SCM. We use the term *mode switching* to denote this ability to change architectural configurations (i.e., to switch between two- and three-party architectures). To model an adaptive architecture, we chose behaviors from the SLP specification (Guttman et al., 1999).

SLP systems are configured by default to operate in three-party mode, switching to two-party mode when SCMs are unavailable. Like Jini, three-party SLP discovery requires that SMs and SUs first discover intermediary SCMs. Upon initiation, a SLP SM or SU enters aggressive discovery, where every 900 s it transmits six probes within a fixed interval of 15 s on a designated aggressive-discovery multicast group. On the other hand, a SLP SCM and SM component commences lazy discovery, where it emits announcements on a designated lazy-discovery multicast group at recommended intervals of 10,800 s (once every 3 h), which we lowered to 120 s in all experiments to provide more consistent behavior in the adaptive and three-party architectures. When operating in three-party mode, SLP SUs and SMs rendezvous through SCMs. After discovery, SLP SMs employ procedures (similar to Jini) to deposit SDs for relevant services on discovered SCMs for a specified TTL, and then to refresh deposited SDs. To make behavior as consistent as possible across our models, we decided to use the same TTLs (on a per experiment basis) for a SD to be cached by a SCM. We denote a specific choice of TTL when describing each experiment (see Section 6). SUs query SCMs for SDs matching their requirements. SCMs process queries, matching SDs against SU requirements, and forward matches to SUs. SUs can cache the response and contact the related SPs to obtain use of the service.

When SLP SUs and SMs fail to detect SCMs, they switch to two-party mode. In two-party mode, a SLP SU both listens for lazy announcements from SMs and transmits the aggressive-discovery six-message probe sequence at 900 s intervals, while SMs listen for probes and respond as appropriate. Upon receiving a lazy announcement or an aggressive-probe response, a SLP SU (in two-party mode) queries the SM for SDs matching its requirements. The SM responds with matching SDs, which the SU caches locally. In the meantime, SUs continue to search for a SCM, using both lazy and aggressive discovery. Upon finding a SCM, SLP requires the SU to switch to three-party mode and to cease direct contact with SMs discovered in two-party mode. All further contact with SMs must take place through SCMs.

3.3. *Modeling consistency maintenance mechanisms*

Service discovery systems include *consistency-maintenance mechanisms* to ensure that changes to critical information about services can be propagated to interested SUs. Critical information could include service availability and capacity, and updates to descriptive information about service capabilities. Discovery systems that we analyzed provide one or both of two consistency-maintenance mechanisms: *polling* and *notification*. We discuss each in turn.

3.3.1. *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; thus, we model the UPnP HTTP `GET` request mechanism to poll the SM to retrieve a SD associated with a specific Uniform Resource Locator (URL). In response, the SM provides a SD containing a list of supported services, including relevant attributes.

Polling in a three-party architecture consists of two independent processes. In one process, a SM sends a request to propagate an updated SD to each SCM on which the SD was originally cached. In Jini, this request takes place through a `ChangeService` message, which causes the SCM to update the cached SD. In SLP, the SM re-registers the SD, which causes the SCM to replace the previously deposited SD with the new version and an updated TTL. In a second process, each SU polls relevant SCMs by periodically issuing a query for a copy of SDs that the SU has previously retrieved and cached. The SCM replies with matching SDs. In Jini, the poll is implemented with a `FindService` request and a `MatchFound` reply; SLP polls (SCMs in three-party mode and SMs in two-party mode) with `SrvRqst` and `SrvReply` messages, respectively. We adopted a 180 s polling interval for all architectures.

3.3.2. *Notification*

Notification requires that updates be transmitted to interested parties immediately after they occur. We model notification only for the two-party and three-party architectures (i.e., not for the adaptive architecture), because the SLP specification that we used does not include notification.

In two-party notification, a SM sends events to a SU that indicates a SD has changed. To receive events about a SD of interest, a SU must first register with the SM for this purpose. We model this procedure using the UPnP subscription mechanism, where the SU sends a `Subscribe` request, and the SM responds by either accepting 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 the TTL for subscriptions in both (the two- and three-party) architectures.

Three-party notification requires a two-step procedure, which we model as specified for Jini. First, SUs must register with SCMs to receive notification about SDs of interest. The SCM registers the notification request for a specified TTL, which may be refreshed. Second, a SM issues a `ChangeService` to propagate a SD update to all SCMs on which the SM has previously deposited the SD. When the SCM receives a `ChangeService` request from a SM for a SD it has cached, the SCM issues a `MatchFound` that propagates the updated SD to all SUs that have registered to receive such notifications.

4. Modeling failure detection and recovery techniques

Interactions among distributed components may be impeded by failures; thus, such components must be prepared to detect failures and take recovery actions. In this section, we review the types of failure that can impede interactions and then we describe selected failure detection and recovery techniques. We explain how we incorporated the techniques into our models.

4.1. Failure types

We classify failures into two general categories: process failures and communication failures. Process failures can be caused by 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 some cases, a node can appear to be present, while being effectively inoperable. Since the effects of node and thread failure are similar, we focus only on node failure in this study, allowing the effects of thread failure to be inferred.

Communication failures can arise due to jamming, or other interference, due to congestion, due to denial of service attacks, 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 subdivide communication failures into three classes: interface failures, message loss, and path failures. 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 dropped, either sporadically or in bursts. Path loss appears as a blocked communication route between two nodes, or areas, in a network. A path can be blocked in one or both directions. Because effects of path failure are similar to interface failure, we studied only interface failure.

4.2. Failure detection and recovery techniques

In service discovery systems, failure detection and recovery responsibilities are divided among three parties: (1) transport protocols, (2) discovery protocols, and (3) applications. The transport protocols support the discovery protocols and the application, while the application also relies on the discovery protocols. We first describe failure detection and recovery provided by transport protocols, such as TCP and UDP. We then discuss heartbeats and soft state – the main detection and recovery techniques implemented by discovery protocols. Subsequently, we discuss remote exceptions and retries, which are the main detection and recovery techniques available to applications and selected discovery processes. We describe how we model these techniques.

4.2.1. Recovery by transport protocols

Discovery protocols and applications use recovery services from three types of transport: (1) unreliable multicast protocols, (2) unreliable unicast protocols, and (3) reliable unicast protocols. We discuss each in turn.

4.2.1.1. Unreliable multicast protocols. Unreliable protocols, whether multicast or unicast, 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 tree. Of course, failures at intermediate points in the tree could result in messages being lost to subsets of receivers. All three systems we studied (UPnP, Jini, and SLP) employ unreliable UDP multicast protocols.

When simulating UDP transmission, our models discard messages lost due to congestion and due to interface failures. During interface failure, the models discard all messages sent from a node with a failed transmitter, as well as all messages inbound for a node with a failed receiver. Neither sender nor receiver learns the fate of lost messages. Since unreliable protocols provide no guarantees, recovery must be provided by mechanisms at a higher layer.

4.2.1.2. Unreliable unicast protocols. Among the systems we studied, both SLP and UPnP use an unreliable unicast protocol. SLP uses unicast UDP to transmit `SrvRqst` messages, used for queries, and to transmit `SrvReg` messages for registrations and registration renewals. To improve reliability, SLP employs two additional procedures. First, SLP issues redundant `SrvRqst` messages; each request is sent four times within a 15 s interval. Second, SLP requires a waiting period (we used 15 s) to listen for a corresponding `SrvRply`. If no `SrvRply` is received within that time, then the message transmission is abandoned and a remote exception (REX) is declared so that a higher layer entity can decide upon an appropriate recovery action. Our SLP models incorporate this behavior.

UPnP uses unicast UDP to send responses to `Msearch` queries. To improve the reliability of these responses, UPnP requires that each UDP message be sent multiple (n) times. In our model, we set $n = 2$.

4.2.1.3. Reliable unicast protocols. Reliable unicast protocols include mechanisms that attempt to ensure message delivery by detecting and re-transmitting lost messages. Of course, the reliability schemes may eventually give up if too many retransmissions are needed (which might indicate node or interface failure). In such cases, the reliable unicast protocol will signal to a higher layer that a message was (probably) not delivered. For example, Jini uses Remote Method Invocation (RMI) over TCP to invoke a method on a remote object and to receive a response, and UPnP uses TCP to submit HTTP requests and receive HTTP responses. Either the RMI layer (in Jini) or the TCP layer (in UPnP) can signal a remote exception (REX).

Our model unifies reliable unicast protocols into one set of procedures that simulate TCP in two phases: connection establishment and data transfer. The connection establishment phase consists of exchanging connection request and response messages. Both connection requests and responses may involve multiple retries before a connection is established. We simulate connection request retries with delays of 6 s, 24 s, and 24 s, before signaling the connection requester with a REX 24 s after the final retry (78 s after the initial request).

Successful connection establishment initiates a data-transfer phase, where the connection requester sends a data request and may await a data response. The data request and response may be subject to retransmissions. We compute a retransmission timeout (RTO) that is roughly the round-trip time, or RTT. We increase the RTO by 25% with each successive retransmission. Retries in the data-transfer phase continue until a time threshold (60 s) is reached, after which the transmission attempt is abandoned. Failure of a data request causes a REX to be issued to the requester. Failure of a data response causes a REX to be issued to both the requester and responder. The requester cannot determine whether a REX was caused by failure to transmit the request or by failure to receive a response. The responder 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.

4.2.2. Recovery by discovery protocols

Components in a discovery system may also learn of failure by listening for recurring messages sent by remote components, much as a heartbeat is monitored to assess patient health. For example, UPnP SMs periodically multicast lazy announcements advertising SDs. Similarly, Jini and SLP SMs periodically refresh SD registrations on SCMs by sending unicast messages, and then listening for responses. Both lazy announcements and registration refresh messages convey *soft* state (or information) – in this case, the SD, which a receiver can cache for a period consistent with the associated TTL. When subsequent heartbeat messages fail to arrive within the TTL, a listener may assume failure of the SM and thus discard cached information about its related SD, effectively eliminating knowledge about existence of the related service.

Our models use a form of soft state that allows SDs for failed components to be discarded and then to be either rediscovered or replaced. For example in our two-party model, once a UPnP SU discards knowledge of a SM and any associated SDs, the SU commences periodic multicast (`Msearch`) queries to search for a new instance of the service. Once the SU regains a SD meeting its requirements, the related queries cease. SLP employs an analogous procedure when operating in two-party mode.

The process is more complicated in three-party situations. Here, failure of refresh messages causes SCMs to discard a service registration. A SU may monitor the status of the SD by periodically polling the SCM. When poll responses indicate the SD is no longer present on the SCM, the SU may then discard its cached copy of the SD. In Jini, SUs may also register with the SCM to be notified when the SCM discards the SD. When receiving such notification, a SU discards its cached copy of the SD and then attempts to find a replacement by querying the SCM for another SD that satisfies its requirements. Meanwhile, a SM for a SD discarded by the SCM might recover after failures are repaired. The SM may rediscover the SCM through aggressive or lazy discovery, and then reregister the lost SD. The SU, if it has not found a replacement, can then receive the original SD by querying the SCM (Jini and SLP) or through notification (Jini).

Table 2
Summary of recovery mechanisms and key parameters

Responsible party	Recovery mechanism	Two-party architecture (UPnP)	Three-party architecture (Jini)	Adaptive Architecture (SLP)
Transport protocols	Multicast UDP	No recovery	No recovery	No recovery
	Unicast UDP	Redundant transmission $n = 2$ No recovery	Not applicable	Redundant transmission $n = 4$ No recovery
	TCP	Issue REX in 78 s	Issue REX in 78 s	Not applicable
Discovery protocols	Heartbeat	SM sends $n(3 + 2d + k)$ lazy announcements of SDs at interval varied by experiment. SU caches SD for TTL varied by experiment, (recommended 1800 s for announcement interval and TTL by UPnP)	SM registers SDs for TTL varied by experiment; SU registers notifications for TTL varied by experiment	SM (in two-party mode only) sends lazy announcements at 120 s interval (recommended 10800 s by SLP); SM registers SDs for TTL varied by experiment
	Soft-state recovery	After purging SD, SU issues aggressive probe (UPnP Msearch) at interval (set to 120 s)	SU and SM issue seven probes (at 5 s intervals) only during startup; SCM issues lazy announcements at interval (120 s)	SU and SM issue 6 probes within 15 s duration during startup and at 900 s interval; SCM sends lazy announcements at 120 s interval (SLP recommends 10800 s)
Application	Ignore REX	SU: *HTTP Get* Poll SM: Notification	SU: *FindService* Poll SCM: Notification	SU:*SrvRqst* Poll (Notification unsupported)
	Retry after REX	SU: *HTTP Get* after discovery retry (180 s with $\leqslant 3$ retries); Registration request and refresh retry (120 s)	SM: depositing or refreshing SD on SCM retry; SU: registering and refreshing notification requests on SCM retry (120 s)	SU:*SrvRqst* after discovery retry (180 s with $\leqslant 3$ retries); SM (three-party mode) depositing or refreshing SD on SCM retry (120 s)
	Discard knowledge	SU purges SD after failure to receive SM announcement within TTL or after 3 retries of *HTTP Get*	SU and SM purge SCM after period of continuous REX (varied by experiment)	Three-party mode: SU and SM purge SCM after period of continuous REX Two-party mode: SU purge SM after period of continuous REX (varied by experiment)

Table 2 summarizes the way in which we model heartbeat and soft state for each of our models. The table indicates values we adopted across all experiments (except as otherwise indicated in the table and discussed in Section 6). In some cases, such as polling and retransmission intervals, we chose relatively short latencies on the assumption that applications would have some expectation of failure and requirement to exhibit robustness. Wherever possible, we based our choices on values recommended in protocol specifications. In doing so, we chose the smallest value from across all the specifications and, to establish consistency, we used that value in all three architecture models. This approach ensured that observed performance differences resulted only from differences in system architecture and protocol.

4.2.3. Recovery by applications

When failure detection leads to a REX, discovery systems generally expect application software to initiate recovery, guided by an application-level retry policy. In our models, depending on the situation, we implement three different policies: (1) ignore the REX, (2) retry the operation for some period, and (3) discard knowledge. The discard strategy, employed following repeated failure of the retry strategy, relies upon discovery mechanisms to recover from failures that are more persistent. These strategies (discussed below) are summarized in Table 2.

4.2.3.1. Ignoring the remote exception. In general, our models ignore any REX received when responding to a request, relying on the requester to retry. A SU can ignore a REX received when issuing a poll (e.g., `FindService`, `SrvRqst`, or HTTP GET) because the poll recurs at an interval. A Jini SCM (three-party model) or UPnP 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 TCP to provide reliability for notifications. Notifications include sequence numbers that allow a receiving node to determine whether or not previous notifications were missed.

4.2.3.2. Retrying the operation. In our models, we retry selected operations in the face of a REX. The UPnP specification separates the operation of discovering a service from obtaining a description of the service (Jini combines these operations). Without a description, a service cannot be used. For this reason, in the UPnP 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 purges the related SD and discards knowledge of the SM. Our three-party models, based on Jini and SLP, also contain 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 recur up to some time bound, after which the SM discards knowledge of the SCM.

4.2.3.3. Discarding knowledge. Both the two-party and three-party models include the possibility that an application can discard knowledge of previously discovered nodes. After discarding knowledge of a SM or SCM, all operations involving that node cease until it is rediscovered, either through lazy or aggressive discovery.

In our UPnP model, SUs discard a SM (and any related SDs) after failure to receive announcements from a SM within a TTL or after three unsuccessful retries of a HTTP GET. In our SLP model (two-party mode), SUs do not discard SMs after failure to receive announcements. We took this decision because the SLP specification does not require SUs to discard a SM when missing a heartbeat.

In our three-party model (based on Jini), a SM or SU deletes a SCM after a period (varied by experiment) of receiving only REXs when attempting to communicate with a SCM. We adopt this behavior because the Jini specification states that a discovering entity *may* discard a SCM with which it cannot communicate. While the SLP specification is silent on these issues, we implemented our SLP model (in both two-party and three-party modes) so that SUs discard SMs after a period (varied by experiment) of continuous REXs. We took this decision to align this behavior among all our models.

5. Experiment methodology

We adopted a common approach to modeling, to experiment design, and to metrics for analysis. Aspects of the approach seem suited to investigation of failure response in other classes of distributed systems. Below, we discuss our approach.

5.1. Model construction

We created simulation models for the three architectures we found. Executable models enabled us to understand collective behavior among distributed components. We based the structure and behavior of our models (recall Section 3) on specifications for UPnP (two-party architecture), Jini (three-party architecture), and SLP (adaptive architecture). Each model comprises a set of components (and relationships among them), interactions (as messages received by components), behavior (as actions taken in response to messages, including generating new messages), and variables (to represent internal state of components). Components communicate via a simulated transport service that represents multicast UDP and unicast UDP and TCP (as explained in Section 4.2.1). The transport service can be impeded by simulated message loss and interface failures. We used Rapide (Luckham, 1996), an architecture description language and accompanying toolset developed at Stanford University, to implement models of Jini and UPnP; for SLP we used SLX, a simulation system developed by Wolverine Software (Henriksen, 1997). We chose to use two different simulation systems in order to establish the generality of our approach. We note that the Rapide system automatically records causal event traces and provides tools to visual and analyze those traces.

5.2. Experiment design

With simulation models in hand, we designed experiments to investigate failure response for selected configurations of components, where each configuration represents a distinct combination of architecture (two-party, three-party, or adaptive), number of deployed SCMs, and choice of behaviors for discovery, consistency maintenance, and recovery. We approached experiment design by focusing on the types of failures (recall Section 4.1) that might interfere with system operation. We decided to consider four failure types: (1) power failure and restart, (2) node failures, (3) interface failures, and (4) message loss. For each failure type, we constructed an application-level scenario to exercise simulated topologies. Our scenarios include: (1) recovering a previously discovered configuration (on restart after power failure), (2) maintaining operational capability in a distributed real-time control application (impeded by failure of nodes hosting needed components), and (3) maintaining consistency of distributed information (when communication is impeded by interface failures or message losses). For three scenarios (node failures, interface failures, and message loss), we subjected each configuration to increasing failure rates, while measuring system response. To focus on fundamental differences in the designs for discovery systems, we excluded a number of possible application-level choices, such as local caching of service descriptions and varying subscription lengths.

5.3. Metrics

To compare failure response among simulated configurations, we defined metrics specific to each scenario. Broadly these metrics fall into three categories: (1) *effectiveness*, which is the ability of a distributed system to exhibit a desired state, expressed as a probability that the state is reached or a proportion of time a system is in the desired state; (2) *responsiveness*, which is the time taken, or latency, to reach the desired state; and (3) *efficiency*, which is the amount of effort, measured by the number of messages, required for a distributed system to complete a scenario. For most combinations of configuration and scenario, we conducted repeated simulations and then we plotted (on the *y*-axis) performance on a metric against increasing failure rate (on the *x*-axis). The graphs also include a table that summarizes performance by averaging a metric across all failure rates; this summarization of the plotted curves gives a quick comparison of relative performance. An exception to this general approach to measurement occurs for the scenario related to restart after power failure, where there is no increasing failure rate. In this case, we simply provide the average and variance of the latency before a configuration is restored. In selected cases, we analyzed event traces to understand how differences in architecture, topology, and behavior contribute to differences in performance.

6. Experiments and results

In this section, we describe our scenarios and exhibit results. For each scenario, we describe the related experiment, delineate the failure model and recovery parameters, define the metrics, display the results and discuss underlying causes. We begin in Section 6.1 with the power-failure-and-restart scenario and then consider in Section 6.2 the distributed real-time control scenario impeded by node failures. Subsequently (in Section 6.3), we discuss the consistency maintenance scenario impeded by communication failures of two types: interface failures and message losses.

6.1. Recovery after power failure

In this experiment, a distributed system establishes an initial configuration in which pairs of SUs and SMs rendezvous, so that each SU obtains one required service. Subsequently, a power failure causes all nodes to crash. Upon power restoration, each SU attempts to rediscover the previously acquired service. This experiment measures the latency until the initial configuration is restored.

6.1.1. Experiment description

This experiment compares three system designs: a two-party model (based on UPnP), a three-party model (based on Jini), and an adaptive model (based on SLP). In the two-party case, the topology (recall Fig. 1) consists of six nodes: three SUs and three SMs. We partition the nodes into three SU–SM pairs that attempt to rendezvous. In the three-party cases (Jini and SLP), the topology (recall Fig. 2) adds three SCMs for a total of nine nodes; however, we use logical partitioning (Jini groups and SLP scopes) so the each SU–SM pair must discover each other through a different SCM; so that a previously discovered configuration may not be rediscovered until all nodes have restarted. We allow all SU–SM pairs to rendezvous, which establishes an initial configuration, and then we simulate a power failure lasting 40 s. We restore power and wait for SUs to rendezvous with the previously discovered SMs. Once the initial configuration is restored, the scenario ends.

Each model includes parameters set to the values indicated in Table 3. The first three rows in Table 3 show parameters unique to specific discovery systems. These parameters include the pattern for aggressive-discovery probes and the interval for lazy-discovery announcements. Jini and UPnP allow SUs to register for notifications; we assume such registrations are lost on node failure. SLP does not allow notifications and thus requires SUs to poll SCMs to discover services. We instantiated the adaptive architecture with two different polling intervals: 31 s as recommended for SLP and 5 s in order to gain early acquisition of services. The fourth row of Table 3 shows parameters for which we selected common values across all models. In particular, note that each node has a restart delay, which in most cases is not defined in discovery specifications. Since the specification for Jini recommends a random delay distributed uniformly between 2 s and 15 s before commencing discovery operations, we decided to assign this same strategy to all of our models in order to eliminate this as a source of difference. The final row of Table 3 lists common transmission and processing delays that we used for each model.

Table 3
Parameters for power failure and restart experiment

Parameter class	Parameter	Value
UPnP protocol parameters	Probe pattern	None used in experiment
	Announce interval	1800 s
	Notification requests	Purge on SM failure
	Polling interval	Not applicable
Jini protocol parameters	Probe pattern	7 Probes 5 s apart
	Announce interval	120 s
	Notification requests	Purge on SCM failure
	Polling interval	Not applicable
SLP protocol parameters	Probe pattern	4 Probes in 15 s
	Announce interval	120 s
	Notification requests	Not applicable
	Polling interval	5 s or 31 s
Common protocol parameters	Registration TTL	30 s
	Total registration duration	100 s
	Node restart delay	2–15 s uniform
Delays used for all models	Transmission delay	1–10 us uniform
	Processing load delay	10–100 us uniform
	Per item processing	10 us (discovery DBs) 100 us (SCM cache)

6.1.2. Metrics

We defined two metrics to compare system performance: *restoration latency* and *efficiency*. Restoration latency measures the elapsed time from restoration of power until the initial configuration is reestablished. Since restoration latency depends upon the starting time of the last system component, we defined *restart delay* to measure the elapsed time from restoration of power until the final system component restarts. We defined efficiency as the total number of messages during restoration latency.

6.1.3. Results

Table 4 presents results, measured over 30 repetitions, for four different configurations. The metrics reveal that for most configurations, restart delay is the dominant

Table 4
Results for power failure and restart experiment

Model variant	Restart delay (s)		Restoration latency (s)		Efficiency (number of messages)	
	Mean	Variance	Mean	Variance	Minimum	Maximum
Two-party	13.07	2.97	15.04	2.97	49	67
Three-party	12.56	2.09	14.76	3.31	70	90
Adaptive (5 s polling interval)	13.13	1.57	16.2	4.25	55	77
Adaptive (31 s polling interval)	13.23	1.22	34.68	65.13	57	100

component of restoration latency; the previous configuration is restored within about 2 s after all nodes have restarted. An exception arises when we configure the adaptive architecture with a 31 s polling interval. Here, the polling interval is the dominant component of restoration latency. This occurs in cases where a related SCM and SU both restart before the SM. Here the SU discovers and queries the SCM for services before the SM can find the SCM and register its service. In this situation, the SU must wait for the 31 s polling interval to elapse for issuing a second, successful query. Reducing the polling interval to 5 s brings restoration latency closer to that exhibited by the other architectures.

Regarding efficiency, Table 4 shows that architectures with more components exchange more messages during a restoration scenario, but those architectures with the same number of components tend to exchange more messages when the scenario takes longer to complete. The three-party architecture proves slightly less efficient than the adaptive architecture because Jini incurs messages related to registration, which SLP does not support.

One final point to note is the slightly better restoration latency of the three-party, as compared with two-party, architecture. This occurs because Jini delivers a service description in one step, concomitant with discovery, while UPnP requires a three-step process: discover the service, get the first part of the service description, and then get the second part of the service description. Should transmission delays increase, this factor would cause even greater difference in restoration latency.

6.2. Service acquisition and maintenance impeded by node failures

In this experiment, we investigate effectiveness and efficiency of service discovery systems in detecting component failure and locating replacements. We model a client for a distributed real-time control application that must discover two types of sensors and an actuator, then monitor sensor readings and control a process. The client has access to a population of sensors and actuators, each running on separate nodes that we allow to fail. The client, sensors, and actuators are supported by a discovery system, represented by configurations of the three architectural variants in our models: two-party (UPnP), three-party (Jini), and adaptive (SLP). Where applicable, the experiment topology may include one or more SCMs, which we also allow to fail. We compare configurations using *functional effectiveness*, measured as the proportion of time that the client possesses an operational set of sensors and actuators required to control the process. We also compare efficiency among configurations by the number of messages exchanged.

6.2.1. Experiment description

Our experiment models a topology that includes one (client) SU and 12 SMs, composed of four instances each of three service types: “fast” sensor, “slow” sensor, and actuator. Fig. 3 illustrates such a topology configured as a two-party architecture and Fig. 4 shows the same topology configured as a three-party architecture (including one to three SCMs). We compare the performance of eight dif-

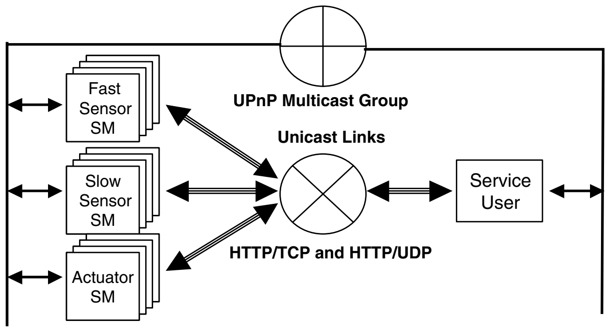


Fig. 3. Two-party service discovery system with one service user and 12 service managers.

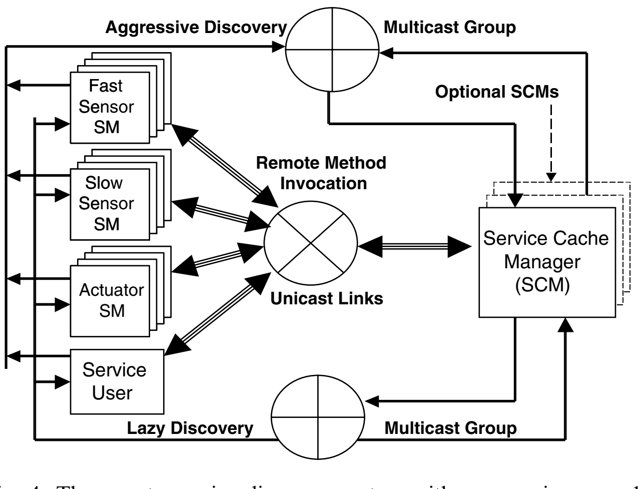


Fig. 4. Three-party service discovery system with one service user, 12 service managers, and up to three service cache managers.

Table 5
Eight configurations compared in node-failure experiment

Configuration	Architecture	Behavior	SCMs
A0	Two-party	UPnP	None
B1	Three-party	Jini	One
B2			Two
B3			Three
C0	Adaptive	SLP	None
C1			One
C2			Two
C3			Three

ferent configurations, enumerated in Table 5. Here, one configuration (**A0**) uses a two-party (UPnP) architecture and one (**C0**) uses an adaptive (SLP) architecture limited to two-party mode, three configurations (**B1**, **B2**, and **B3**) use a three-party (Jini) architecture, and three configurations (**C1**, **C2**, and **C3**) use an adaptive, three-party (SLP) architecture.

To establish initial conditions, we exercise each configuration until discovery completes and the SU acquires one service of each of the three service types. We then fail nodes according to the failure model described below. In order to focus exclusively on failure detection and recovery processes, we do not allow the SU to cache backup services, so at any time the SU holds at most one SD for each service type. After activation, a "fast" sensor transmits a reading every two seconds and a "slow" sensor transmits a reading every 30 s. The SU invokes the actuator after receiving an appropriate combination of readings from a "fast" and "slow" sensor. We select actuation times randomly from a uniform distribution with a mean of 60 s, provided the SU receives the required sensor readings. When the SU holds one SD for a service of each type ("fast" sensor, "slow" sensor, and actuator) and when each of those services is operational, then the application is considered *functional*. If the SU lacks SDs for one or more service type or if one or more of the SDs held by the SU describes a service instance that is not operational, then the application is considered *non-functional*. When non-functional, the SU client must first detect what services have failed and then initiate recovery procedures to discover replacements. During each experiment repetition, we accumulate the periods when the client is non-functional as well as the time required for failure detection and recovery. We also record message counts of the underlying service discovery system for the experiment duration.

6.2.2. Failure model

During the experiment duration T_D, each SM node (and SCM node in three-party configurations) fails randomly and independently, although at least one service of each type always remains active. We take this decision to provide a reasonable test of the service discovery system's capability to recover and restore the application to a functional state. We let λ be the node failure rate that varies from 0% to 80% in 10% increments (though no failures occur when $\lambda = 0$). The mean time to node failure is $t_{MF} = (1 - \lambda) \cdot T_D$. Node failure times are randomly chosen from a "stepped" normal distribution with three steps: a 0.15 probability of failure before $t_{MF} - 0.2t_{MF}$, a 0.7 probability of failure between $t_{MF} - 0.2t_{MF}$ and $t_{MF} + 0.2t_{MF}$, and a 0.15 probability of failure between $t_{MF} + 0.2t_{MF}$ and $2t_{MF}$. Failure times are distributed uniformly within each step. When a node fails, affected services become unavailable for a time, selected from three failure duration classes, each with a different probability and duration. Short failures occur with a probability of 0.1 for a fixed (135 s) duration; intermediate failures occur with a probability of 0.7 for a duration selected uniformly on the interval [180, 300] s, long failures occur with a probability of 0.2 selected uniformly on the interval [480, 600] s.

6.2.3. Failure recovery techniques

Table 6 gives common and configuration-specific parameters for failure recovery techniques we used in this experiment. We chose parameters that enable the SU to respond quickly to failure of remote services and to find replacements as soon as possible. We describe the recovery techniques employed in our model: first at the discovery level and then at the application level.

6.2.3.1. Discovery-level recovery. For the two-party (UPnP) architecture, we use a heartbeat and soft-state strategy, choosing a TTL of 600 s for refreshing cached SDs. If not refreshed within the TTL, the SU purges the SD and commences periodic (120 s) Msearch queries to find a replacement service. When we model SLP in two-party mode, the SU both listens for lazy announcements (120 s) from SMs and periodically issues multicast queries for SMs (900 s) to find replacements. In three-party configurations (both Jini and SLP), we model heartbeat monitoring through service registration refreshes on SCMs, choosing a refresh interval of 30 s for slow sensors and actuators and 300 s for fast sensors. If refreshes are missed, the SCM purges the SD. In the three-party architecture, a SU that discovers a SD through a SCM polls that SCM every 180 s to learn if the SD has been purged; if so, the SU assumes failure of the related service and also purges the SD. In both three-party and adaptive architectures, SUs and SMs search for SCMs by listening for lazy announcements (120 s).

6.2.3.2. Application-level recovery. Across all models, we adopt an identical application-level recovery policy: upon failure to receive a scheduled sensor reading (every 2 s for fast sensors and 30 s for slow sensors) the SU immediately purges the related SD and commences search for a replacement. Similarly, failure to receive a response to an actuation attempt within 20 s causes the SU to purge the related SD and to commence search. A similar policy applies to detecting failed SCMs. If a SM does not receive a response when attempting to refresh a service registration, the SM assumes

Table 6
Recovery parameters for node-failure experiment

	Configuration	Parameter	Value
Discovery-level recovery	Behavior for two-party UPnP configuration **A0**	Announce interval	600 s (lowered from recommended value)
		SU purges SD	At TTL expiration (600 s)
		Msearch query interval	120 s
	Behavior for two-party SLP configuration **B0**	Multicast query interval	120 s
	Behavior for three-party Jini and SLP configurations **B1**, **B2**, **B3**, **C1**, **C2**, **C3**	Refresh interval	30 s for slow sensors and actuators 300 s for fast sensors
		SCM purges SD	Immediately after a missed refresh
		SU-SCM query interval	180 s
		SU purges SD	Immediately after learning SD is unavailable
Application-level recovery	All configurations	Sensor interval	2 s for fast sensors 30 s for slow sensors
		SU purges SD	Immediately after missed sensor reading and after failing to receive an actuation response within 20 s
		SM or SU purges SCM	20 s after failure to receive response to request

that the SCM has failed and begins searching for a replacement. Similarly, if a SU does not receive a response to a SCM query, the SU purges the SCM and begins to search.

6.2.4. Metrics

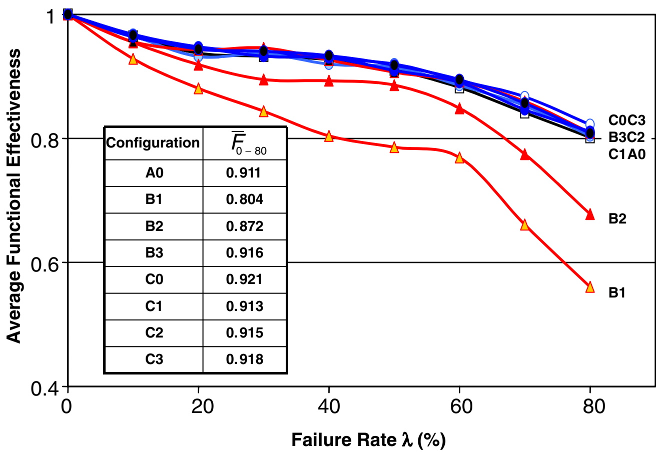
We define T_{NF} as accumulated time during which a client application is in a non-functional state. We compute the proportion of T_{D} that a client application is in a functional state, or the client's functional effectiveness, by the ratio $F = (T_{\mathrm{D}} - T_{\mathrm{NF}})/T_{\mathrm{D}}$. We compute the average functional effectiveness of a configuration at a particular failure rate λ for n experiment repetitions as

$$\overline{F}_{\lambda} = \frac{\sum_{i=1}^{n}(\langle T_{\mathrm{D}}\rangle_i - \langle T_{\mathrm{NF}}\rangle_i/\langle T_{\mathrm{D}}\rangle_i)}{n}$$

We measure T_{NF} as follows. As indicated, a client that has become non-functional first incurs a delay before detecting the failure. We call this delay *detection latency*. After detecting a non-functional state, the client may incur some delay while restoring required services. We call this delay *recovery latency*. Detection latency commences when a SM fails but the SU holds a SD provided by the SM. Once the SU discards the SD, or the SM recovers, detection latency ends. Recovery latency begins after the SU purges a SD for a failed service and commences search. Recovery latency ends when the SU finds a SD matching its needs. During periods when a client incurs either detection or recovery latency or both (the states can overlap), the client is non-functional, and we accumulate such periods in T_{NF}.

6.2.5. Results

For each of the eight configurations in Table 5, we set $T_{\mathrm{D}} = 1800$ s and executed 60 repetitions for each failure rate λ. Fig. 5 shows average functional effectiveness $\overline{F}_{\lambda}$ for each configuration as λ increases. Fig. 5 also includes a table that shows the summary statistic $\overline{F}_{0\text{–}80}$, which is $\overline{F}_{\lambda}$ averaged across all values of λ for each configuration.



Configuration	$\overline{F}_{0-80}$
A0	0.911
B1	0.804
B2	0.872
B3	0.916
C0	0.921
C1	0.913
C2	0.915
C3	0.918

Fig. 5. Comparing average functional effectiveness $\overline{F}_{\lambda}$ for different configurations in response to increasing rate of node failures, where at least one SM of each type is operational (60 repetitions per data point). The table gives the $\overline{F}_{0\text{–}80}$, or functional effectiveness averaged across all values of λ for each configuration.

The results show that six of the eight configurations have similar curves for $\overline{F}_{\lambda}$ and a $\overline{F}_{0\text{–}80}$ of over 0.9. The three-party configurations with one SCM (**B1**) and two SCMs (**B2**) perform less well, because as λ rises, the incidence of failure of the single SCM in **B1** and concurrent failure of both SCMs in **B2** increases. With no SCM to query for services, the SU remains non-functional. Adding a third SCM (**B3**) reduces the probability of concurrent SCM failure sufficiently to raise $\overline{F}_{0\text{–}80}$ to a level comparable with other configurations. The adaptive architecture achieves a comparable $\overline{F}_{0\text{–}80}$ even with two or fewer SCMs, because when no SCMs can be found, the SU immediately switches to two-party mode to discover the available SMs. In the discussion below, we provide more detail on the effectiveness of these configurations by considering their comparative detection and recovery latencies.

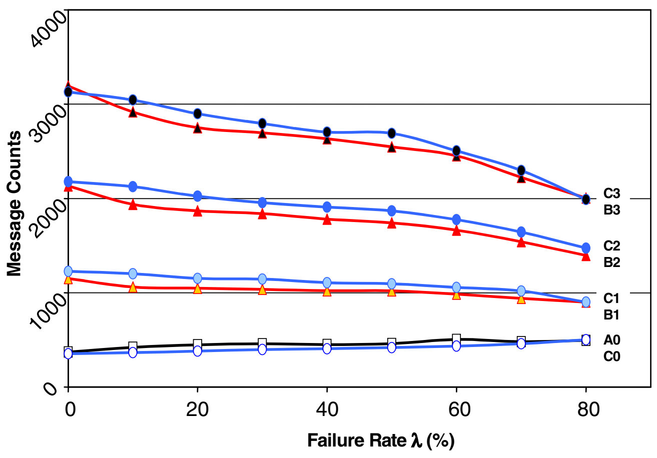


Fig. 6. Comparing message counts for different configurations in response to increasing rate of node failures where at least one SM of each type is operational (60 repetitions per data point).

As revealed in Fig. 6, efficiency varies markedly among the configurations. The two-party configurations **A0** and **C0** are notably more efficient than any three-party configuration. This occurs in part because more messages are needed for SUs and SMs to rendezvous through SCMs. These messages include heartbeats by the SCMs, registration and refresh of SDs by SMs, and polls of SCMs by the SU. In the three-party and adaptive architectures, differences in protocol also influenced efficiency. For equivalent configurations, the three-party architecture (**B1**, **B2**, and **B3**) proves more efficient than the adaptive architecture (**C1**, **C2**, and **C3**). This occurs, because in the former, Jini SCMs send lazy announcements at 120 s intervals, while Jini SUs and SMs employ aggressive search only at start-up. However, in the adaptive architecture, both SLP SCMs *and* SMs announce every 120 s, while SUs and SMs repeat a six-probe aggressive search sequence at regular intervals (900 s). We believe that with equivalent underlying behaviors, adaptive and three-party architectures would exhibit similar efficiency when configured with an equal number of SCMs.

One additional point is worth noting. In the two-party configurations (**A0** and **C0**), the message-count curves have increasing slope as λ increases, because the SU must search more frequently for replacement services. Note, however, that three-party configurations have message-count curves with decreasing slope as λ increases. The rate of message exchange decreases because SCMs fail more frequently and remain down for longer periods as λ rises, thus reducing the number of opportunities for SD refresh messages and SCM heartbeats.

6.2.6. Discussion

While three-party configurations with three SCMs (**B3** and **C3**) yield comparable functional effectiveness to two-party configurations (**A0** and **C0**), our experiment reveals quite different underlying causes. Fig. 7a–c display similar non-functional time (T_{NF}) under increasing failure rate

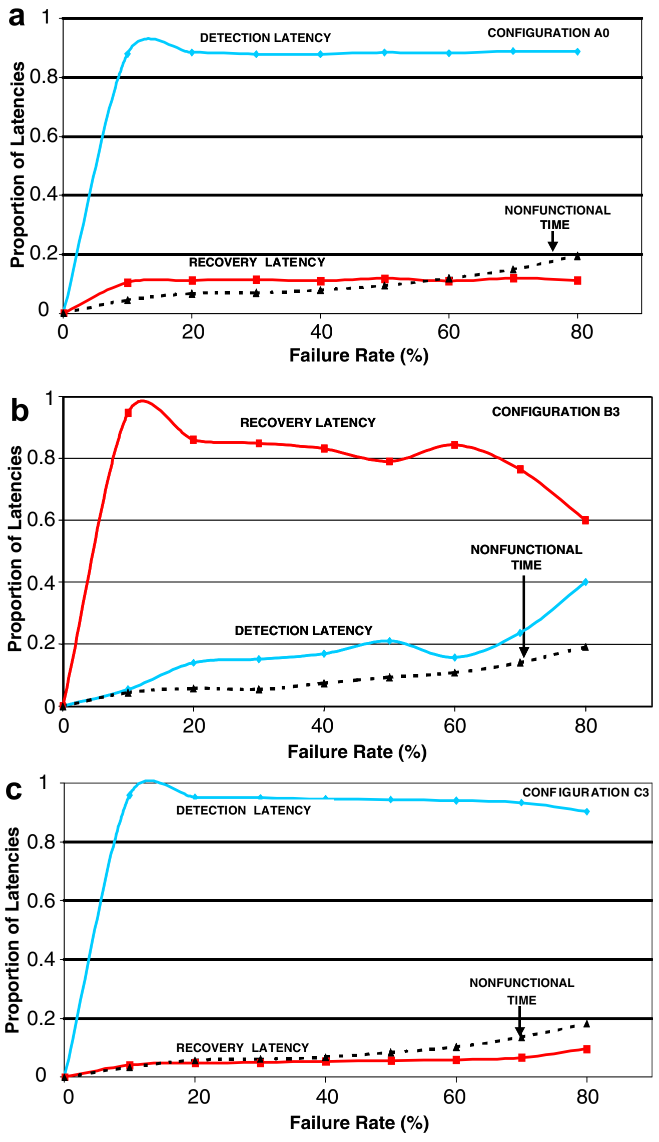


Fig. 7. Detection and recovery latencies of various configurations as a proportion of nonfunctional time (60 repetitions per data point). Decomposition of nonfunctional time into detection and recovery latency for configuration: (a) **A0** (b) **B3** and (c) **C3**.

for configurations **A0**, **B3**, and **C3**. The figures also decompose T_{NF} into the proportions attributable to detection latency and recovery latency. In the two-party configuration, reported in Fig. 7a, about 90% of T_{NF} accrues while waiting to detect a failure; recovery occurs quickly. Analysis of execution traces showed most failures were detected through missed sensor readings or REXs received in response to failed actuations. In the three-party configuration, shown in Fig. 7b, the situation is different. Here, the largest component of T_{NF} is recovery latency. Execution traces for the three-party architecture show incidence of concurrent failure of all SCMs rising steadily with increasing λ. With no SCMs available, the SU is unable to find replacements for failed services until a SCM (1) recovers, (2) is discovered by the SU and SMs, (3) accepts registrations from available SMs, and (4) responds to queries from the SU. These factors dramatically increased the proportion of T_{NF} attributable to recovery latency. This trend is more marked with fewer SCMs (not shown here). In the adaptive configuration, as displayed in Fig. 7c, over 90% of T_{NF} is again detection latency. Here, upon detecting failure, the SU switches to two-party mode when no SCMs can be found; thus, avoiding the delay incurred in waiting for a SCM to recover. Hence, the detection and recovery behavior of the adaptive configuration appears quite similar to the two-party configuration, which is also reflected in the similarity of Fig. 7a and c.

6.3. Consistency maintenance impeded by communication failures

In this experiment, we investigate effectiveness and efficiency of service discovery systems in maintaining consistency of information replicated throughout a distributed system. We model five clients (SUs) that each discover the same service manager (SM) and obtain a copy of the service description (SD) managed by the SM. Subsequently, the SM updates its local copy of the SD, creating an inconsistency with the SDs replicated to the SUs. We measure the probability that each SD will receive an updated copy of the SD prior to a deadline, the latency incurred in receiving the updated SD, and the number of messages exchanged to convey the update. We consider effects from two types of communication failure, interface failures and message losses, which could impede dissemination of the updated SD. We also compare two alternate consistency maintenance mechanisms: polling (recall Section 3.3.1) and notification (recall Section 3.3.2), which are supported by selected discovery systems.

6.3.1. Experiment description

We compare performance of nine configurations, as enumerated in Table 7. One configuration (**A0p**) uses a two-party (UPnP) architecture (see Fig. 8) with a polling regime to maintain consistency. Another configuration (**A0n**) combines the same architecture with notification. Four configurations (**B1p**, **B1n**, **B2p** and **B2n**) use a three-party (Jini) architecture (see Fig. 9) with one or two SCMs and polling or notification. Three configurations (**C0p**, **C1p** and **C2p**) use an adaptive (SLP) architecture (with zero, one, or two SCMs) and polling (SLP does not include a notification mechanism).

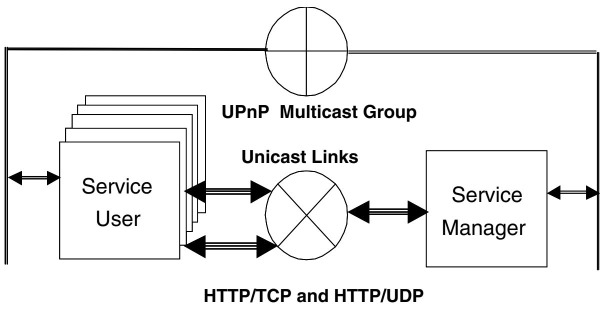


Fig. 8. Two-party service discovery system deployed in a six-node topology: five service users and one service.

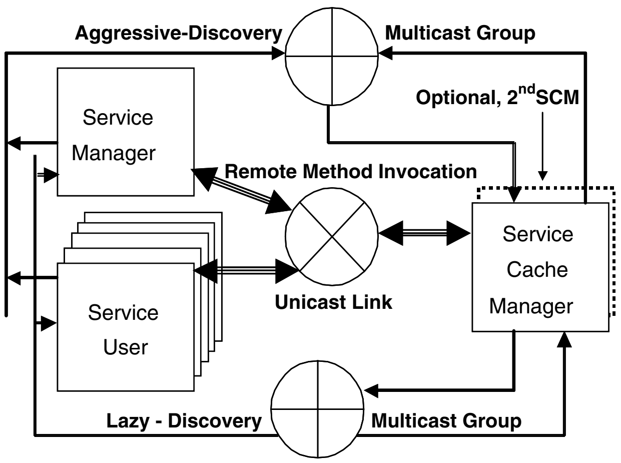


Fig. 9. Three-party service discovery system deployed in a seven- or eight-node topology: five service users, a service manager, and one or two service cache managers.

To establish initial conditions, we set aside an interval, up to time t_Q, for all SUs to discover the SM and obtain the SM's SD. We then activate interface failures or message loss according to the appropriate failure model described below. In addition, we establish a deadline t_D by which the change must propagate to all SUs, and then choose a time, randomly distributed on the uniform interval $[t_Q, t_D/2]$, to introduce a change in the SD on the SM. Here, we set $t_Q = 100$ s and $t_D = 5400$ s. Each experiment aims to

Table 7
Nine configurations compared in communication-failure experiments

Configuration	Architecutre	Behavior	Consistency-maintenance mechanism
A0p	Two-party	UPnP	Polling
A0n	Two-party	UPnP	Notification (with notification registration on SM)
B1p	Three-party (one SCM)	Jini	Polling (with service registration on SCM)
B1n	Three-party (one SCM)	Jini	Notification (with service registration and notification registration on SCM)
B2p	Three-party (two SCMs)	Jini	Polling (with service registration on SCM)
B2n	Three-party (two SCMs)	Jini	Notification (with service registration and notification registration on SCM)
C0p	Adaptive (no SCMs)	SLP	Polling
C1p	Adaptive (one SCM)	SLP	Polling (with service registration on SCM)
C2p	Adaptive (two SCMs)	SLP	Polling (with service registration on SCM)

restore consistency between the changed SD held by the SM and the cached copies of the SD held by the SUs. We recorded the time of change to the SD on the SM, the latency required to propagate the update to each SU prior to t_D (or failure to do so) and the number of messages exchanged.

6.3.2. Failure models

We conducted separate experiments for interface failure and message loss. Table 8 summarizes relevant parameters for each failure model.

6.3.2.1. Interface failure. In the interface-failure experiment, we let λ be the interface failure rate. During the experiment, each node suffers an interface failure at a time, randomly distributed on the uniform interval $[t_Q, t_D - (t_D \cdot \lambda)]$. When activating each interface failure, there is an equal likelihood that the transmitter, receiver, or both fail. Once activated, each failure remains in effect for the duration of $t_D \cdot \lambda$, after that the failure is remedied. During a failure interval, no messages are sent from a node with a failed transmitter, and a node with a failed receiver does not receive messages. For each configuration simulated, we varied λ from 0 to 90% in increments of 5%.

6.3.2.2. Message loss. In the message-loss experiment, we let λ be the message-loss rate. For each attempt to transmit a message, whether on a reliable or unreliable channel, a uniform random real number is selected from the unit interval $[0, 1]$. If the number is less than λ, the message is discarded. Loss of a message sent on a reliable channel stimulates a retransmission after an appropriate timeout. We varied λ as in the interface-failure experiment.

6.3.3. Failure recovery techniques

We model recovery techniques at three levels: transport protocols, discovery protocols, and application. Recovery techniques for the transport protocols are described in Section 4.2.1. Table 9 shows the recovery techniques and related parameters we adopted for the discovery and application levels.

6.3.3.1. Discovery-level recovery. In the two-party (UPnP) architecture, we use a heartbeat and soft-state strategy where SUs discarded SDs not refreshed within a TTL (of 1800 s). To enable rediscovery of SMs (and SCMs, where applicable) we adopt a discovery behavior consistent with the specific protocol (UPnP, Jini, or SLP) being modeled. In all configurations (except **A0p**, which does not employ registration), we chose the same TTL (of 1800 s) after which registrations would be discarded if not renewed. For REXs received in response to registration or refresh attempts, to ad-hoc queries, or to change-service operations, the retries occur at intervals of 120 s (but only up to a maximum of 540 s). To comply with the Jini and UPnP specifications, there are no retries after a REX when attempting to issue notifications.

6.3.3.2. Application-level recovery. For configurations (**A0p**, **B1p**, **B2p**, **C0p**, **C1p**, and **C2p**) that use polling, we set the polling interval to 180 s. In (UPnP) configurations (**A0p** and **A0n**), SUs discard a SD after (HTTP GET) queries

Table 8
Parameters for interface failure and message loss models

Failure	Parameter	Value
Interface failure	Failure incidence	Once per run for each node
	Failure scope	Transmitter, reciever, or both with equal likelihood
	Failure duration	5% increments of 5400 s from 0% to 90%
Message loss	Failure incidence	Each transmission may fail with probability equal to message loss rate from 0% to 90%
	Failure scope	Individual message transmission
	Failure duration	Individual message transmission

Table 9
Key model parameters for communication-failure experiments

	Configuration	Parameter	Value
Discovery level recovery	**A0p** and **A0n** (UPnP)	Announce interval	1800 s
		Msearch query interval	120 s
		SU purges SD	At TTL expiration
	B1p, **B1n**, **B2p** and **B2n** (Jini)	Probe interval	5 s (7 times)
		Announce interval	120 s
	C0p, **C1p**, and **C2p** (SLP)	Probe interval	Variable (4 probes in 15 s)
		Announce interval	900 s
	A0n, **B1p**, **B1n**, **B2p**, **B2n**, **C1p** and **C2p**	Registration TTL	1800 s
	A0n, **B1p**, **B1n**, **B2p**, **B2n**, **C0p**, **C1p** and **C2p**	Time to retry after REX	120 s
Application-level recovery	**A0p**, **B1p**, **B2p**, **C0p**, **C1p** and **C2p**	Polling interval	180 s
	A0p, **A0n**, and **C0p**	SU purges SD	After 540 s with only REX
	B1p, **B1n**, **B2p**, **B2n**, **C1p** and **C2p**	SM or SU purges SCM	After 540 s with only REX

to the SM result in nothing but REXs for a total of 540 s. In other configurations, SUs discard a SCM after receiving nothing but REXs over 540 s while attempting to interact with the SCM.

6.3.4. Metrics

We evaluate update effectiveness, responsiveness, and efficiency. Update effectiveness measures the probability that a change to a SD will propagate to a given SU before the deadline t_D. We let n be the number of repetitions of an experiment, m be the number of SUs in a topology, and t'_{ij} be the time that an updated SD is propagated to SU$_j$, $1 \leqslant j \leqslant m$, in experiment repetition i, $1 \leqslant i \leqslant n$. Then, we define update effectiveness for the failure rate λ over n repetitions as

$$U_\lambda = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m} chg_{ij}}{n \cdot m}$$

where

$$chg_{ij} = \begin{cases} 1 & \text{if } t'_{ij} < t_D, \\ 0 & \text{otherwise} \end{cases}$$

defines whether a change in a SD was propagated to the jth SU during the ith repetition (i.e., 1 if true, 0 if false).

Update responsiveness measures the latency in propagating the SD update. We let t'_i be the time the SD change occurred on the SM in experiment repetition i. Update responsiveness $\widetilde{R}_\lambda$ is the median of all $1 - p_{ij}$ at a particular value of λ where

$$p_{ij} = \frac{t'_{ij} - t'_i}{t_D - t'_i}$$

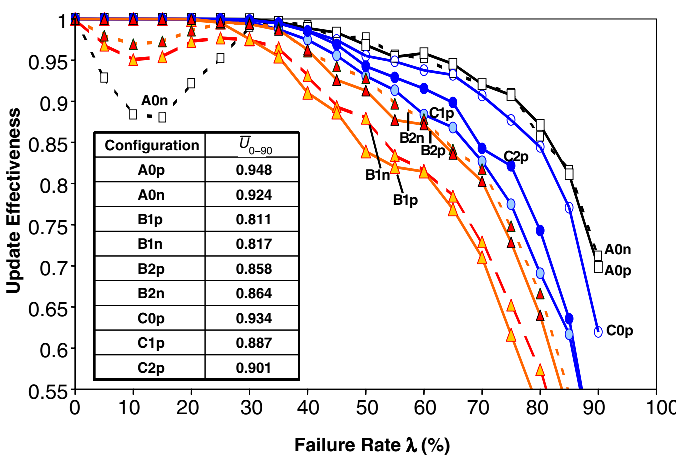
is the proportion of time required to propagate an update to the jth SU in the ith repetition at λ.

Update efficiency measures the effort required to (attempt to) maintain consistency. Analysis of our experiment configurations revealed a minimum number of messages, x, that must be sent to propagate a change to all SUs. This minimum $(x = 7)$ occurred for the three-party configuration with notification and one SCM (**B1n**)[2]. We define update efficiency based on the ratio of x to the actual number of messages observed. We let y be the number of messages sent while attempting to propagate a change from the SM to the SUs in a given repetition. Then, for n number of experiment repetitions, we define average update efficiency at a particular failure rate λ as

$$\overline{E}_\lambda = \frac{\sum_{i=1}^{n}(x/y_i)}{n}.$$

6.3.5. Interface-failure results

For each configuration in Table 7, we executed $n = 1000$ repetitions at each interface-failure rate λ. Fig. 10 shows update effectiveness U_λ for the configurations as λ



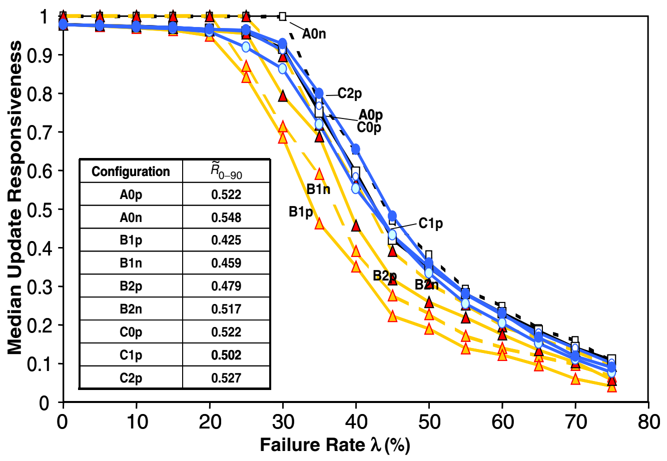
Configuration	$\overline{U}_{0\text{–}90}$
A0p	0.948
A0n	0.924
B1p	0.811
B1n	0.817
B2p	0.858
B2n	0.864
C0p	0.934
C1p	0.887
C2p	0.901

Fig. 10. Comparing update effectiveness (U_λ) for different configurations in response to increasing rate of interface failures (1000 repetitions per data point). The table gives $\overline{U}_{0\text{–}90}$, or U_λ averaged across all values of λ for each configuration.

increases. The figure also includes a table with mean update effectiveness $\overline{U}_{0\text{–}90}$, which is U_λ averaged across all values of λ for each indicated configuration. Overall, these results show that a two-party architecture, or an adaptive architecture that has a two-party mode, provides superior effectiveness to a three-party architecture (at least given topologies limited to one or two SCMs). This occurs because each updated SD must propagate over only one channel (SM to SU) in two-party cases, but over two channels (SM to SCM and SCM to SU) in three-party cases. For both three-party and adaptive architectures, $\overline{U}_{0\text{–}90}$ improves with the number of SCMs due to the reduction in the incidence of joint failure of both channels. We note that polling yields better effectiveness than notification. For example, when comparing three-party polling with one SCM (**B1p**) against three-party notification with one SCM (**B1n**), the advantage of polling appears as λ exceeds 35% because when notifications fail, SD updates are propagated by recovery mechanisms, which activate only after some delay. On the other hand, polling persists with retries after receiving a REX. We note that configurations using notification also exhibit anomalous behavior when λ is in the range [5, 25]%; we discuss the reasons for this below in Section 6.3.7.

Fig. 11 shows median update responsiveness $\widetilde{R}_\lambda$ for all configurations as λ increases. Generally, the ranking of architectures for responsiveness is similar to effectiveness. Where employed, notification exhibits better responsiveness than polling, which incurs increased latency from the 180 s polling interval. Fig. 11 also shows a steep drop-off in $\widetilde{R}_\lambda$ for all configurations as λ increases beyond the [20, 30]% range, where failures prevent initial propagation of the updated SD, forcing invocation of recovery actions that cannot succeed until paths are restored. Thus, even though some configurations achieved effectiveness of over 0.9 as λ reaches 70% (see Fig. 10), responsiveness for all configurations approaches zero. Three-party configura-

[2] Recall that the two-party (UPnP) architecture requires a multiple-message exchange to convey SDs.



Configuration	$\widetilde{R}_{0-90}$
A0p	0.522
A0n	0.548
B1p	0.425
B1n	0.459
B2p	0.479
B2n	0.517
C0p	0.522
C1p	0.502
C2p	0.527

Fig. 11. Comparing median update responsiveness ($\widetilde{R}_\lambda$) for different configurations in response to increasing rate of interface failures (1000 repetitions per data point). The table gives $\widetilde{R}_{0-90}$, which is $\widetilde{R}_\lambda$, averaged across all values of λ for each configuration.

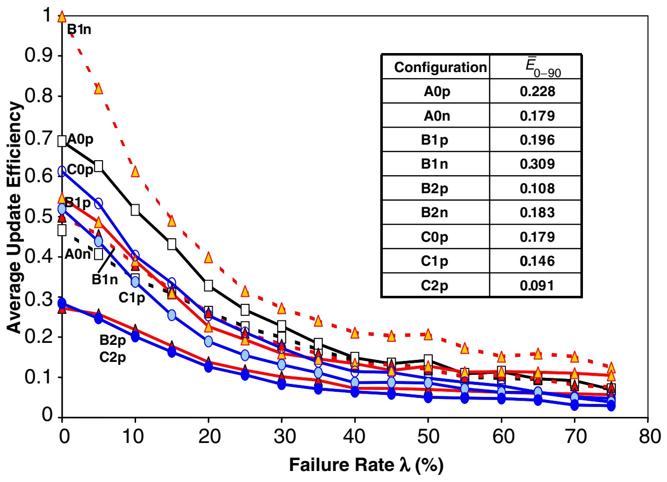
tions experience longer delays at high values of λ as paths to SCMs become increasingly unavailable.

Fig. 12 shows average efficiency $\overline{E}_\lambda$ for experiment configurations as λ increases. The table included in Fig. 12 shows $\overline{E}_{0-90}$, which is $\overline{E}_\lambda$ across all values of λ for each indicated configuration. Here, $\overline{E}_\lambda$ declines for all configurations as λ increases. This reflects a rising number of messages generated when recovery strategies are invoked more frequently as λ rises. Configurations using more SCMs are less efficient (but more effective) than similar configurations with fewer SCMs. The adaptive architecture appears less efficient than the three-party architecture with an equivalent number of SCMs for the reasons described above in Section 6.2.5. Again, we expect the use of equivalent underlying behaviors would yield comparable efficiencies.

Some other points seem worth noting. The three-party configurations using notification (**B1n** and **B2n**) are more efficient than similar configurations using polling (**B1p** and **B2p**) because in Jini each SU poll to a SCM involves a request followed by a reply, while a Jini SCM notification is a single message. However, for $\lambda < 40\%$, two-party (UPnP) notification (**A0n**) appears less efficient than two-party polling (**A0p**). This occurs because when UPnP notifications are lost, recovery strategies must often be used, thus prolonging the time to propagate the updated SD and increasing message counts.

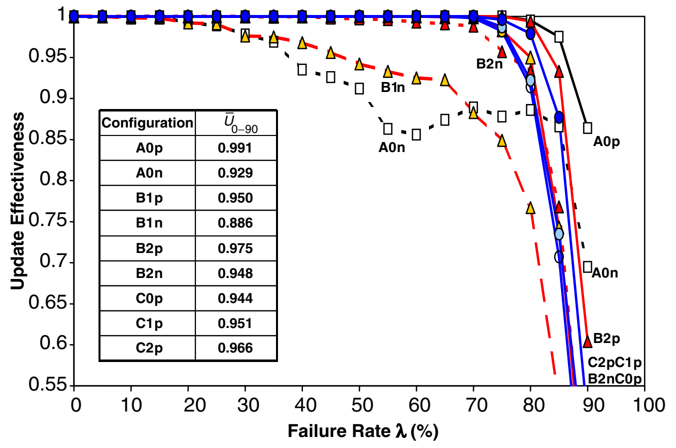
6.3.6. Message-loss results

For each configuration in Table 7, we executed $n = 200$ repetitions at each message-loss rate λ. Fig. 13 shows update effectiveness U_λ for the configurations as λ increases. Fig. 13 also includes a table that shows $\overline{U}_{0-90}$ across all values of λ for each indicated configuration. Overall, these results show that most configurations provide an effectiveness of 0.95 or better until λ exceeds 80%. Overall, effectiveness under message loss conditions is higher than under interface failure conditions. This is because interfaces fail for protracted periods at higher values of λ, increasing the probability that channels remain blocked until t_D, so updates never get through. In contrast, message loss affects only individual transmissions, allowing recovery strategies more opportunities to propagate the update before t_D. Polling continues to yield better effectiveness than notification. The two-party configuration with polling (**A0p**) achieves a mean effectiveness of 0.99, due to the combined advantages of using polling with just two parties (which requires transiting one channel rather than two). We note that the two-party configuration with notification (**A0n**) and the three-party notification with one SCM (**B1n**) exhibit anomalous behavior and reduced effectiveness as λ surpasses 20%; we discuss the reasons



Configuration	$\overline{E}_{0-90}$
A0p	0.228
A0n	0.179
B1p	0.196
B1n	0.309
B2p	0.108
B2n	0.183
C0p	0.179
C1p	0.146
C2p	0.091

Fig. 12. Comparing average update efficiency ($\overline{E}_\lambda$) for different configurations in response to increasing rate of interface failures (1000 repetitions per data point). The table gives $\overline{E}_{0-90}$, which is $\overline{E}_\lambda$ averaged across all values of λ for each configuration.

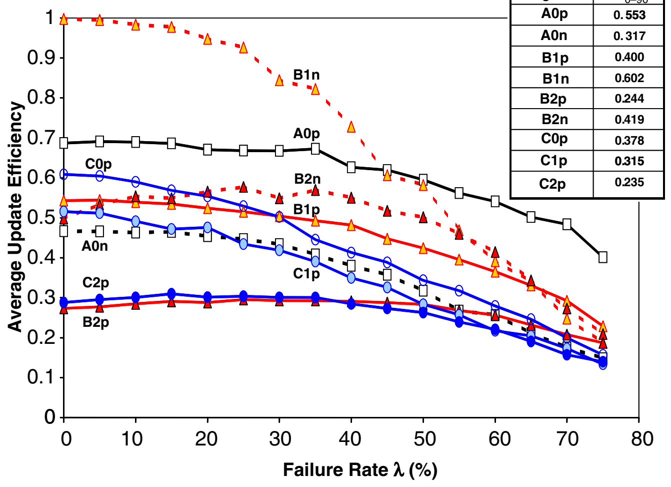


Configuration	$\overline{U}_{0-90}$
A0p	0.991
A0n	0.929
B1p	0.950
B1n	0.886
B2p	0.975
B2n	0.948
C0p	0.944
C1p	0.951
C2p	0.966

Fig. 13. Comparing update effectiveness (U_λ) for different configurations in response to increasing rate of message loss (200 repetitions per data point). The table gives $\overline{U}_{0-90}$, or U_λ averaged across all values of λ for each configuration.

for this below in Section 6.3.7. Responsiveness (not shown here) exhibits a steep decline after $\lambda > 80\%$, compared with $\lambda > 30\%$ for interface failure. The higher responsiveness under message loss conditions occurs for the same reasons as higher effectiveness. Under message loss, notification also continues to provide better responsiveness than polling.

Fig. 14 shows average efficiency $\overline{E}_\lambda$ for experiment configurations as λ increases and includes a table for $\overline{E}_{0-90}$ for each configuration. As in the case of effectiveness and responsiveness, all configurations prove more efficient under message loss conditions than under interface failure for the reasons given above. The better efficiency is also reflected in the overall more gradual decline in the message loss efficiency curves. Otherwise, the general ordering of efficiencies for the various configurations appears similar under both interface failure and message loss. We note the reduced efficiency of the two-party (UPnP) notification (**A0n**) above $\lambda = 20\%$ in comparison with two-party polling (**A0p**). In **A0n**, efficiency suffers from cases where notifications are lost and recovery procedures are required to propagate the update (taking more time and requiring more messages). The combination of lost notifications and use of recovery also causes a sharp decline in the efficiency of the three-party notification with a single SCM (**B1n**), which at low values of λ, generates the fewest messages to propagate updates. Another exception is the three-party configuration using notification with two SCMs **(B2n)**, which exhibits increasing efficiency over the failure rate range $[5, 35]\%$ and overtakes the three-party configuration using polling with one SCM (**B1p**). This counterintuitive result occurs because in some repetitions, lost messages cause the SM or SUs to discover only one of the two SCMs; thus, messages that would normally be duplicated to both SCMs are not.



Configuration	$\overline{E}_{0-90}$
A0p	0. 553
A0n	0. 317
B1p	0.400
B1n	0.602
B2p	0.244
B2n	0.419
C0p	0.378
C1p	0.315
C2p	0.235

Fig. 14. Comparing average update efficiency ($\overline{E}_\lambda$) for different configurations in response to increasing rate of message loss (200 repetitions per data point). The table gives $\overline{E}_{0-90}$, which is $\overline{E}_\lambda$ averaged across all values of λ for each configuration.

6.3.7. Discussion

The notification mechanism included in UPnP and Jini (and other distributed systems) proved unexpectedly ineffective at disseminating updates under certain conditions. Foremost, under low interface-failure rates (in the range $[5, 30]\%$) our results exhibit saw-tooth phenomena for configurations using notification. The dip is most pronounced (nearly 15%) for the two-party (UPnP) configuration (**A0n**) and less pronounced (around 5%) for the three-party (Jini) configurations (**B1n** and **B2n**). In the two-party case, analysis of execution traces showed 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). Since neither UPnP nor Jini require notification senders to retry after a REX, updated information must be disseminated through a recovery mechanism. At low failure rates, a notification can be lost to an interface failure, which is repaired prior to the next announcement or registration-refresh attempt. Under such conditions, recovery mechanisms are not invoked and the SU does not obtain an updated SD. Polling proves more effective because the SU checks periodically (180 s intervals) and persistently for updated information and retrieves the SD when indicated.

A similar sequence of events occurs in the three-party case, but the effects are more modest. The three-party configurations require a SM to first propagate a change to a SCM. Failure to propagate a change results in a REX that causes the SM to retry the change for up to 540 s, during which time the interface failure may be repaired. If still unconfirmed after 540 s, the SM purges the SCM and initiates aggressive discovery. After rediscovering the SCM, the SM propagates the change, and the SCM then notifies registered SUs. Even with this redundancy, there still is some chance that a SU receiver is blocked and thus unable to receive notification. The redundancy does, however, increase the probability that an updated SD reaches a SU.

Notification (as specified for UPnP and Jini) also appears less effective under message loss. Lack of application-level retries to deliver notices leads to significant decline in update effectiveness above $\lambda = 20\%$. This appears for the relevant two-party (UPnP) configuration (**A0n**) and three-party (Jini) configuration (**B1n**), both of which use notification. Above $\lambda = 20\%$, the incidence of undelivered notifications increases and, unless recovery is stimulated, the updated SD is not disseminated. In configuration **A0n**, as λ exceeds 60 %, lost registration-refresh requests trigger recovery procedures with increasing frequency, which causes propagation of the updated SD when a registration is reestablished. This process slightly improves and then maintains effectiveness within the failure rate range [60, 80]%, causing this curve to echo the saw-tooth feature in the update effectiveness curve for **A0n** under interface failure. Above $\lambda = 80\%$, lost messages effectively close the channel, and effectiveness collapses for all configurations.

For the three-party configuration (**B1n**), loss of change requests (from the SM) as well as registration refreshes (from the SM and SUs) also stimulate recovery procedures that partly compensate for lost notifications. When a second SCM is added (configuration **B2n**) update effectiveness improves because the SM now has two paths through which to disseminate updates to SUs.

7. Conclusions

Overall, we found designs for first-generation discovery systems can be robust under difficult failure environments. Across all experiments, most configurations exhibited an effectiveness of better than 0.9 in obtaining services or propagating updates for failure rates approaching (often exceeding) 80%. Configurations proved ineffective only when all essential nodes failed or were unreachable, or when recovery actions were not activated (as occurred in response to lost update notifications). Similarly, extensive delays in propagating updates depended on the duration of path outages.

For our scenarios and metrics, two-party configurations (or three-party configurations that could adapt to two-party mode) appeared more robust than three-party configurations (where robustness improved with the number of replicated directories). Deploying three directory replicas yielded robustness equal to two-party configurations. In tradeoff, increasing the number of directory replicas lowers system efficiency by increasing the number of messages exchanged. In most cases, we found the adaptive architecture with one directory achieved robustness comparable to other configurations, while providing better efficiency than configurations with replicated directories.

To disseminate updates, we found polling more effective than notification. Our polling regime used persistent retries, while our notification regime depended only on reliable transport protocols, falling back to alternate recovery mechanisms when notifications could not be delivered. The alternate recovery mechanisms were not always activated at lower failure rates. This anomaly appeared in effectiveness plots for configurations using notification. Notification generally conveyed updates with less delay than polling. In the two-party architecture, polling was more effective, so scenarios tended to end earlier and require fewer messages.

Beyond our methodology and comparisons, we identified and discussed the most significant design and configuration decisions that influence robustness and efficiency in first-generation discovery systems. We showed how available architectural alternatives, as well as choices for consistency maintenance and recovery strategies, lead to robustness-efficiency tradeoffs. We also showed how faulty assumptions regarding recovery strategies could unexpectedly degrade robustness and efficiency. The information provided should convey a better understanding of failure behavior in existing discovery systems, allowing potential users to configure deployments for high robustness at low cost. The discussions presented here could also help to improve designs for future discovery systems.

Acknowledgements

We received generous funding support from Susan Zevin, as acting director of the NIST Information Technology Laboratory, Douglas Maughan, as manager of the Defense Advanced Research Projects Agency (DARPA) Fault-Tolerant Networks Program, John Salasin, as manager of the DARPA program in Dynamic Assembly for System Adaptability, Dependability and Assurance, and James Puffenbarger of the Advanced Research and Development Activity (ARDA).

References

Allard, J., Chinta, V., Gundala, S., Richard, G., 2003. Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability. In: Proceedings of the 2003 International Symposium on Applications and the Internet (SAINT 2003), Orlando, FL, (January), pp. 268–275.

Arnold, K. et al., 1999. The Jini Specification, Version 1.0. Addison-Wesley.

Bettstetter, C., Renner, C., 2000. A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol. In: Proceedings of the Sixth EUNICE Open European Summer School: Innovative Internet Applications, Open EUNICE 2000, Twente, Netherlands, September.

Specification of the Bluetooth System, Core, Version 1.1, vol. 1, the Bluetooth SIG, Inc., 2001.

Bowers, K., Mills, K., Rose, S., 2003. Self-adaptive Leasing for Jini. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), Fort Worth, TX, (March), pp. 539–542.

Bushmitch, D., Lin, W., Bieszczad, A., Kaplan, A., Papageorgiou, V., Pakstas, A., 2004. A SIP-Based Device Communication Service for OSGi Framework. In: Proceedings of the 2004 IEEE Consumer Communications And Networking Conference, Las Vegas, NV, (January), pp. 453–458.

Castro, M. et al., 2002. One Ring to Rule them All: Service Discovery and Binding in Structured Peer-to-Peer Overlay Networks. In: The Proceedings of the Tenth ACM SIGOPS European Workshop, ACM, Saint-Émilion, France, (September).

Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C., 2001. Grid Information Services for Distributed Resource Sharing. In: Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10), San Francisco, CA, (August), pp. 181–194.

Czerwinski, S. et al., 1999. An Architecture for a Secure Service Discovery Service, Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCom '99), Seattle, WA, (August), pp. 24–35.

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

Dabrowski, C., Mills, K., Elder, J., 2002a. Understanding Consistency Maintenance in Service Discovery Architectures During Communications Failure. In: Proceedings of the 3rd International Workshop on Software Performance, Rome, Italy, (July), pp. 168–178.

Dabrowski, C., Mills, K., Elder, J., 2002b. Understanding Consistency Maintenance in Service Discovery Architectures In Response to Message Loss. In: Proceedings of the 4th International Workshop on Active Middleware Services, Edinburgh, United Kingdom, (July), pp. 51–60.

Dabrowski, C., Mills, K., Rukhin, A., 2003. Performance of Service-Discovery Architectures in Response to Node Failure. In: Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, NV, (June), pp. 95–104.

Dabrowski, C., Mills, K., Quirolgico, S., 2005. A Model-based Analysis of First-Generation Service Discovery Systems, Special Publication 500-260, National Institute of Standards and Technology.

Frolund, S. et al., 2000. Building Dependable Internet Services with E-speak, Hewlett Packard Laboratories Technical Report HPL-2000-78.

Guttman, E., Kempf, J., 1999. Automatic Discovery of Thin Servers: SLP, Jini and the SLP-Jini Bridge. In: Proceedings of the 25th Annual Conference of the IEEE Industrial Electronics Society (IECON 99), Volume 2, San Jose, CA, December, pp. 722–727.

Guttman, E., Perkins, C., Veizades, J., Day, M., 1999. Service Location Protocol, Vol. 2, Internet Engineering Task Force (IETF), RFC 2608.

Henriksen, J., 1997. An Introduction to SLX^{TM}. In: Proceedings of the 1997 Winter Simulation Conference, ACM, Atlanta, GA, December, pp. 559–566.

Specification of the Home Audio/Video Interoperability (HAVi) Architecture, Version 1.1, HAVi, Inc., 2001.

Hsiao, H., King, C., 2002. Neuron – A Wide-Area Service Discovery Infrastructure. In: Proceedings of the International Conference on Parallel Processing (ICPP '02), Vancouver, British Columbia, (August), p. 455.

Iamnitchi, A., Foster, I., 2001. On Fully Decentralized Resource Discovery in Grid Environments. In: Proceedings of an IEEE International workshop on Grid computing, Denver, CO, November.

Joseph, S., 2002. NeuroGrid: Semantically Routing Queries in Peer-to-Peer Networks. In: Proceedings of the International Workshop on Peer-to-Peer Computing, Pisa, Italy, May.

JXTA v2.0 Protocols Specification, Sun Microsystems, 2004. Available from: <http://spec.jxta.org/v1.0/docbook/JXTAProtocols.html>.

Luckham, D., 1996. Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events. Available from: <http://anna.stanford.edu/rapide>.

Miller, B., Pascoe, R., 1999. Mapping Salutation Architecture APIs to Bluetooth Service Discovery Layer, Version 1.0, Bluetooth SIG white paper, July.

Mills, K., Dabrowski, C., 2003. Adaptive Jitter Control for UPnP M-Search. In: Proceedings of 2003 IEEE International Communications Conference, Anchorage, AK, May.

Mills, K., Rose, S., Quirolgico, S., Britton, M., Tan, C., 2004. An Autonomic Failure-Detection Algorithm. In: Proceedings of the 4th International Workshop on Software Performance (WoSP 2004), San Francisco, CA, January, 79.

O'Driscoll, G., 2000. Essential Guide to Home Networking Technologies. Prentice-Hall Trading Company.

Olivier, B., 2000. Jini: a platform for building adaptive integrated learning environments, Report from the Centre for Learning Technology (CeLT), University of Wales Bangor, United Kingdom, December.

Pascoe, R., 1999. Salutation Architectures and the newly defined service discovery protocols from Microsoft and Sun: How does the Salutation Architecture stack up, Salutation Consortium white paper.

Pascoe, R., 2001. Building Networks on the Fly. IEEE Spectrum 38 (3), 61–65.

Ponnekanti, S., Fox, A., 2003. Application-Service Interoperation without Standardized Service Interfaces. In: Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PerCom 2003), Fort Worth, TX, (March), pp. 30–39.

Rekesh, J., 1999. UPnP, Jini and Salutation – A look at some popular coordination framework for future network devices, Technical Report, California Software Lab.

Richard, G., 2000. Service Advertisement and Discovery: Enabling Universal Device Cooperation. IEEE Internet Computing 4 (5), 18–26.

Richard, G., 2002. Service and Device Discovery: Protocols and Programming. McGraw-Hill.

Rose, S., Bowers, K., Quirolgico, S., Mills, K., 2003. Improving Failure Responsiveness in Jini Leasing. In: Proceedings of the Third DARPA Information Survivability Conference and Exposition (DISCEX-III 2003), Volume 2, Washington, DC, (April), pp. 103–105.

Salutation Architecture Specification, Version 2.0c, Salutation Consortium, June 1999.

Sameh, A., El-Kharboutly, R., 2004. Modeling Jini-UPnP Bridge using Rapide ADL. In: Proceedings of the IEEE/ACS International Conference on Pervasive Services (ICPS'04), Beirut, Lebanon, (July), p. 237.

Sundramoorthy, V., Speelziek, M., van de Glind, G., Scholten, J., 2004. Service Discovery with FRODO. In: 12th IEEE International Conference on Network Protocols (ICNP), Berlin, Germany, (October), pp. 24–27.

Sundramoorthy, V., 2006. At Home In Service Discovery, PhD dissertation, University of Twente, Netherlands.

Tan, C., Mills, K., 2005. Performance Characterization of Distributed Algorithms for Replica Selection in Distributed Object Systems, Accepted for Fifth International Workshop on Software Performance (WoSP 2005), Palma de Mallorca, Spain, July.

UDDI Technical White Paper, published by the members of uddi.org, September 2000.

Universal Plug and Play Device Architecture (UPnP), Version 1.0, Microsoft, Inc., 2000.

Verma, D. et al., 2003. SRIRAM: A scalable resilient autonomic mesh. IBM Systems Journal 42 (1), 19–28.

Yu, M., Taleb-Bendiab, A., Reilly, D., Omar, W., 2003. Multi-Standard Service Interoperation Protocol through Polyarchical Middleware. In: Proceedings of the PostGraduate Networking Conference (PGNet), Liverpool, United Kingdom, (June), pp. 143–148.