Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2007-031 June 4, 2007

CAPRI: A Common Architecture for Distributed Probabilistic Internet Fault Diagnosis

George J. Lee

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CAPRI: A Common Architecture for Distributed Probabilistic Internet Fault Diagnosis

by

George J. Lee

B.S., University of California, Berkeley (2000)
S.M., Massachusetts Institute of Technology (2003)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

© Massachusetts Institute of Technology 2007. All rights reserved.

Author . . . . . .
Department of Electrical Engineering and Computer Science
May 23, 2007

Certified by . . . . . .
David D. Clark
Senior Research Scientist, Computer Science and Artificial Intelligence Laboratory
Thesis Supervisor

Accepted by . . . . . .
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

CAPRI: A Common Architecture for Distributed Probabilistic Internet Fault Diagnosis

by

George J. Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis presents a new approach to root cause localization and fault diagnosis in the Internet based on a Common Architecture for Probabilistic Reasoning in the Internet (CAPRI) in which distributed, heterogeneous *diagnostic agents* efficiently conduct diagnostic tests and communicate observations, beliefs, and knowledge to probabilistically infer the cause of network failures. Unlike previous systems that can only diagnose a limited set of network component failures using a limited set of diagnostic tests, CAPRI provides a common, extensible architecture for distributed diagnosis that allows experts to improve the system by adding new diagnostic tests and new dependency knowledge.

To support distributed diagnosis using new tests and knowledge, CAPRI must overcome several challenges including the extensible representation and communication of diagnostic information, the description of diagnostic agent capabilities, and efficient distributed inference. Furthermore, the architecture must scale to support diagnosis of a large number of failures using many diagnostic agents. To address these challenges, this thesis presents a probabilistic approach to diagnosis based on an extensible, distributed **component ontology** to support the definition of new classes of components and diagnostic tests; a **service description language** for describing new diagnostic capabilities in terms of their inputs and outputs; and a **message processing procedure** for dynamically incorporating new information from other agents, selecting diagnostic actions, and inferring a diagnosis using Bayesian inference and belief propagation.

To demonstrate the ability of CAPRI to support distributed diagnosis of real-world failures, I implemented and deployed a prototype network of agents on Planetlab for diagnosing HTTP connection failures. Approximately 10,000 user agents and 40 distributed regional and specialist agents on Planetlab collect information from over 10,000 users and diagnose over 140,000 failures using a wide range of active and passive tests, including DNS lookup tests, connectivity probes, Rockettrace measurements, and user connection histories. I show how to improve accuracy and cost by learning new dependency knowledge and introducing new diagnostic agents. I also show that agents can manage the cost of diagnosing many similar failures by aggregating related requests and caching observations and beliefs.

Thesis Supervisor: David D. Clark
Title: Senior Research Scientist, Computer Science and Artificial Intelligence Laboratory

Acknowledgments

Many people have helped me over the course of my Ph.D. research. First of all, I thank my thesis advisor Dave Clark for granting me the freedom to explore new ideas while keeping me focused on the important research questions. Dave has helped me broaden my perspective and think about the larger contributions and implications of my research. Dave's insights and feedback have greatly improved the clarity and rigor of my thesis. I also thank my thesis committee members Karen Sollins, Howie Shrobe, and Dina Katabi for their support and guidance. I thank Karen Sollins for her insightful questions, countless suggestions, and helping me see how my work fits in to the bigger picture. I am grateful to Howie Shrobe for his friendly support and encouragement. Howie provided many valuable suggestions about how to improve and evaluate the output of diagnosis in CAPRI. Dina Katabi's probing questions and valuable feedback helped me better understand the contributions of my thesis and the advantages of probabilistic inference in CAPRI.

Many others have also helped me with the ideas in my thesis research. I thank my officemate Steve Bauer for generously listening to my half-baked ideas and for countless thought-provoking discussions over the years. Steve's insights, suggestions, and criticisms have been invaluable in my research. I am grateful to Peyman Faratin for introducing me to many important people, papers, and subjects related to my thesis work. My many discussions with Peyman have helped me better understand the strengths and weaknesses of my work and the design decisions I made. I thank Lalana Kagal for her feedback and suggestions about how to use ontologies in CAPRI. My conversations with David Sontag improved my understanding of various techniques for probabilistic logic and inference. I thank Garrett Wollman for sharing some of his real-world network fault diagnosis knowledge and experience.

I thank John Wroclawski for his guidance in my early years as a Ph.D. student. His keen insights and criticisms over the years have helped me develop my critical thinking skills and better identify weaknesses in my own work. I thank Rob Beverly and Ji Li for their many helpful comments and feedback about my research. I am grateful to many others for their friendship and feedback over the years, including Joanna Kulik, Mike Afergan, Simson Garfinkel, Arthur Berger, Xiaowei Yang, Ben Leong, Indranil Chakraborty, Yu-Han Chang, and Kai Shih. I thank Becky Shepardson and Lisa Gaumond for all their hard work and assistance.

Finally, I thank my parents for their endless encouragement, support, and guidance.

Contents

List of Figures

Chapter 1

Introduction

Often when network failures occur in the Internet, users and network administrators have difficulty determining the precise location and cause of failure. For example, a failure to connect to a web server may result from faults in any one of a number of locations; perhaps the local network is misconfigured, a failure may have occurred along the route to the server, or perhaps the server's network card has failed. To a large extent the difficulty of diagnosis results not from a lack of tools or techniques for diagnosis, but from the challenge of efficiently coordinating the exchange of diagnostic observations, beliefs, and knowledge across multiple administrative domains in an unreliable and dynamic network. Accurate diagnosis involves accurately determining the status of many interdependent network *components*, including processes such as DNS lookups and devices such as Ethernet switches. Proper diagnosis of failures in such components may require performing multiple imprecise diagnostic tests and intelligently inferring the cause of failure from their results.

Unfortunately, gathering accurate information for diagnosis in the Internet is not easy. Network topologies and the status of links may change rapidly and render previously collected information useless. New classes of applications and devices may become available that require new techniques to diagnose. Researchers may develop new diagnostic tests and methods for inference. Network failures may prevent the collection of certain observations and diagnostic information. Excessive diagnostic probing and communication may introduce distortions or cause additional failures. Adding to the challenge, often collecting the data necessary to diagnose a failure requires coordination among administrators in multiple autonomous systems (ASes) that may have different capabilities and conflicting interests. Due to such challenges, today the diagnosis of network failures requires slow and tedious manual data collection, inference, and communication among human network administrators. As the size, complexity, and diversity of the Internet increases, so will the difficulty of performing such manual diagnosis.

To address the challenges of distributed diagnosis, this thesis presents a new approach to fault diagnosis based on a Common Architecture for Probabilistic Reasoning in the Internet (CAPRI) in which distributed, heterogeneous *diagnostic agents* with different capabilities and goals efficiently conduct diagnostic tests and communicate observations, beliefs, and knowledge to probabilistically infer the cause of failures in the Internet. This thesis takes a completely different approach to the problem of fault diagnosis compared to previous work. Rather than developing a distributed system specifically designed for diagnosing particular

types of failures as previous researchers have done, the primary objective of this thesis is to develop a general, extensible framework for fault diagnosis not specific to the diagnosis of any particular type of failure and not restricted to any particular set of diagnostic tests. Instead, CAPRI provides a common architecture for cooperation among multiple distributed specialists, allowing experts to improve the system by adding new diagnostic tests and new dependency knowledge in the future. The generality of the CAPRI architecture allows the sharing of diagnostic information among agents that use a wide range of both new and existing methods for fault diagnosis, including both well-known active measurements such as DNS lookups and traceroutes as well as passive inference of web server status from historical connectivity observations. Like the Internet Protocol (IP) that glues together heterogeneous link layer technologies to provide a common interface for higher-level communication protocols and applications, CAPRI enables the communication of diagnostic information from various sources to provide a common platform for distributed diagnosis. Chapter 2 describes the similarities and differences between CAPRI and previous research in related areas.

A common and extensible architecture for fault diagnosis in the Internet has the potential to catalyze innovation, encouraging the development of new diagnostic tools and technologies. As agents join the system and the number and variety of CAPRI agents increases, new diagnostic capabilities and applications will emerge and the power and accuracy of the system will improve. To facilitate such innovation, CAPRI provides modular interfaces for communicating diagnostic information, encouraging specialization, reuse, information hiding, and composition of information. Researchers may develop new agents with special knowledge, technology, or resources for diagnosing a particular set of components. CAPRI enables such specialist agents to join the system and share their diagnostic capabilities with other agents. If the information that a specialist provides is useful, other agents may reuse the same information to diagnose many types of failures. Agents may also hide specialized information from others to help manage complexity and improve scalability, reducing the amount of information that other agents need to perform diagnosis. For example, an agent may advertise the capability to diagnose all types of DNS lookup failures without revealing information about all the other agents it needs to contact to perform that diagnosis. CAPRI also enables the composition of information from multiple agents to produce new information. For example, an agent for diagnosing HTTP connection failures might have no special capabilities of its own and simply combine observations, beliefs, and dependency knowledge from other agents to produce its diagnosis. Like the online encyclopedia Wikipedia[1] which brings together distributed specialists to share information and build upon each other's knowledge to produce a shared repository of valuable information, CAPRI provides a framework for specialist agents to contribute their diagnostic capabilities and build upon the capabilities of other agents. Chapter 3 precisely defines the problem of fault diagnosis and describes the elements of the CAPRI architecture.

The strength of the CAPRI architecture comes from its generality and extensibility to support diagnosis using a wide range of diagnostic tests, knowledge, and communication patterns, but achieving such generality requires overcoming several challenges including representation of diagnostic information, description of diagnostic capabilities, commu-

[1] http://www.wikipedia.org/

nication of diagnostic information, and the selection of diagnostic actions. This thesis describes how CAPRI addresses each of these challenges while providing the generality, extensibility, and scalability to support the introduction of a large number of new diagnostic agents and capabilities.

Firstly, since diagnosis requires the communication of diagnostic information among multiple diagnostic agents, CAPRI must provide agents with a common language for the extensible representation and communication of network component and diagnostic test class definitions. This language must support a wide range of diagnostic information and allow researchers to define new types of information to support new types of diagnostic tests. CAPRI addresses this challenge by representing component and diagnostic test classes and properties using an extensible, distributed **component ontology**, enabling agents to define new component classes and properties to support new components and new diagnostic tests. Chapter 4 describes the representation of diagnostic information in more detail.

Secondly, because each agent may have the capability to perform a different set of diagnostic tests and may not know about the existence of all other agents in the network, CAPRI must provide agents with a common language for describing and looking up diagnostic capabilities. This language must allow new agents to join the system and advertise their capabilities so that existing agents can take advantage of the new capabilities offered. For scalability, this language must also control the complexity of routing diagnostic information among a large number of diagnostic agents. CAPRI addresses this challenge by providing a common **service description language** that enables agents to describe diagnostic capabilities in terms of their inputs and outputs. This language allows other agents to dynamically compute the value of services offered by other agents and select actions in a general way without domain-specific knowledge. This service description language also promotes scalability by allowing agents to aggregate multiple specialized services to reduce the number of services each agent needs to know about. Chapter 5 describes service advertisement and lookup in more detail.

Thirdly, distributed diagnosis requires a common protocol for exchanging diagnostic observations, beliefs, and knowledge. In order to support diagnosis using new information produced by new agents, this protocol must allow an agent to combine information collected from multiple sources to build a dependency model without domain-specific knowledge. CAPRI addresses this challenge by providing agents with a common **message exchange protocol** that enables agents to express diagnostic information in terms of *observations*, probabilistic *beliefs* and *likelihoods*, and probabilistic *dependency knowledge* in a general way. This protocol allows agents to construct component graphs to represent information they have about the network and probabilistic failure dependency graphs for inferring the cause of failure in a general way. Chapter 6 describes the protocol that agents use for exchanging diagnostic information.

Fourthly, for extensibility to support new diagnostic capabilities offered by new agents, agents need a general procedure for computing the value of new services and deciding what actions to take. In addition, this procedure must scale to support the diagnosis of a large number of failures using many agents. To address this challenge, CAPRI provides agents with a general **message processing procedure** for dynamically computing the value of available actions based on available diagnostic information, service descriptions, and probabilistic dependency knowledge. This procedure enables agents to dynamically con-

struct failure dependency graphs from a component graph and a dependency knowledge base for performing probabilistic inference with incomplete information. In addition, for scalability to support the diagnosis of a large number of failures this procedure manages the long-term average probing and communication costs of diagnosis by aggregating data and requests and propagating diagnostic information according to an aggregation-friendly overlay topology. Chapter 7 describes this procedure in more detail.

This thesis primarily considers the diagnosis of reachability failures, in which a user cannot access a particular resource or service in the Internet. This thesis focuses on reachability failures because such failures occur relatively frequently and are difficult to diagnose using information collected from any single point in the network, but can often be accurately diagnosed by combining data collected from multiple points in the network and performing inference. One can extend this architecture to deal with other types of failures such as performance failures, but such failures are outside the scope of this thesis. This thesis addresses only the problem of automated fault diagnosis and does not address other parts of the management process such as automated configuration and repair. Other challenges of distributed fault diagnosis include privacy and security, agent discovery, trust, and incentives, but addressing such challenges is not the focus of this thesis.

This thesis evaluates CAPRI according to three criteria: the ability to support distributed diagnosis among heterogeneous agents, extensibility to support new diagnostic tests and knowledge, and scalability to control the cost of diagnosing large numbers of failures using many agents. Note that the purpose of my evaluation is not to show that diagnostic agents can diagnose any particular types of failures using any particular set of diagnostic tests; rather, the purpose of my evaluation is to gain insight into the benefits and tradeoffs of diagnosis using CAPRI.

To evaluate the capabilities of the CAPRI architecture, I developed a prototype network of agents on Planetlab for diagnosing real-world HTTP connection failures for end users. Chapter 8 describes the design and implementation of this prototype network. In my experiments, agents collect information from over 10,000 users and diagnose over 140,000 failures over two months. These experiments demonstrate distributed diagnosis among eight different types of diagnostic agents, including approximately 10,000 user agents and 40 regional and specialist agents on Planetlab. I show how the extensibility of the CAPRI ontology and communication protocol enables agents to improve accuracy and cost by taking into account new diagnostic knowledge. I show how agents can use the message processing procedure provided by CAPRI to dynamically compute the value of new services and select diagnostic actions. I show how agents in this network can improve the scalability of diagnosing large numbers of failures and reduce the probing and communication costs of diagnosis by aggregating related requests and caching observations and beliefs. This experiment also demonstrates the benefits of probabilistic inference, including the ability to indicate the confidence of a diagnosis, compute the value of new actions, and learn new dependency knowledge. Chapter 9 presents the detailed results of the experimental evaluation.

The creation of a common architecture for fault diagnosis in the Internet opens up many new avenues of future research. Chapter 10 concludes with a discussion of the contributions of this thesis research and areas of future work.

Chapter 2

Related Work

This thesis presents a new approach to fault diagnosis in the Internet based on a common architecture for communicating diagnostic information that allows the introduction of new diagnostic agents and new diagnostic capabilities. This thesis draws upon many areas of related work, including tools for Internet fault diagnosis, architectures for agent coordination, the representation of diagnostic information and services, root cause localization, dependency analysis, reasoning and logic with incomplete information, and probabilistic reasoning.

Unlike previous research in network management, probabilistic inference, machine learning, and distributed systems, however, this thesis takes a broader, long-term, wide-area networking approach to distributed fault diagnosis in which extensibility, robustness to failure, communication costs, scalability, heterogeneity, incomplete information, and conflicting interests all matter. In the Internet we cannot assume, for example, that all diagnostic agents know about all components in the network. The Internet comprises thousands of separate administrative domains, and no single entity can possibly know the entire structure of the Internet at any given moment in time. Communicating diagnostic information to all users affected by a failure may involve communication among possibly thousands or millions of hosts; we must consider the cost of such communication. Each diagnostic agent may have different capabilities, and the set of available agents may change as new agents join or network failures occur. In addition, the dynamic and unpredictable nature of the Internet means that diagnostic tests may produce unreliable, incomplete, or out-of-date information. Furthermore, any lasting architectural solution to this problem must be flexible and extensible to accommodate the introduction of new classes of components and diagnostic technologies. This chapter divides related work into four broad categories: distributed systems, diagnostic tools, representation, and fault diagnosis.

2.1 Distributed systems

CAPRI agents perform diagnosis in a distributed manner. The challenges of distributed diagnosis include coordination among distributed diagnostic agents and controlling the cost of communication.

2.1.1 Architectures for agent coordination

CAPRI is an architecture for coordinating diagnosis between multiple agents, each with different resources, capabilities, and administrators. This architecture differs from most other architectures for agent coordination in that agents exchange diagnostic information according to an extensible ontology that enables them to reason about new classes of components and new properties, as well as dynamically request additional information from other agents when necessary. Agents exchange high-level problem-solving knowledge such as component status and conditional probability tables, and not just low-level domain factual knowledge about network observations such as link bandwidth or round-trip packet delay. This approach is based on the idea of the Knowledge Plane, a common infrastructure for communicating data about the network that enables reasoning about high-level concepts such as policies, configuration, and behavioral models [12].

One obstacle to the adoption of a common architecture for sharing network data is that network administrators see little incentive to make such data available to others. Today, network administrators do not share much management information with administrators in other domains for security and policy reasons. Calvert and Groffioen argue, however, that network administrators might benefit from sharing of policy and configuration data, possibly in aggregated and anonymized form, because it can simplify tasks such as fault diagnosis [5].

Networking researchers have developed various architectures for exchanging network observations among agents for distributed diagnosis. Wawrzoniak, et al. developed a distributed knowledge base called Sophia that allows agents to make queries about the state of a distributed network using a declarative logic language [93]. The iPlane is another distributed system for collecting and providing network observations [63]. Both Sophia and the iPlane are designed for the exchange of low-level domain factual knowledge of network observations such as link bandwidth and loss rates, whereas agents in CAPRI exchange more high-level problem-solving information such as dependency knowledge and beliefs.

Performing diagnosis using such higher-level knowledge and beliefs improves the modularity and extensibility of diagnosis. A researcher may introduce a new diagnostic agent that can produce the same diagnostic information as another agent using a new technique. Other agents may then take advantage of the information provided by the new agent without knowing the low-level details about the new technique. In addition, communicating dependency knowledge enables agents to take advantage of new types of diagnostic tests about new classes of network components when such information becomes available. Chapter 4 discusses the advantages of communicating knowledge and beliefs in more detail.

Thaler and Ravishankar describe an architecture for distributed fault diagnosis using a network of experts [89]. Each component in the system has a corresponding expert that can determine whether that component has failed and what other components may be affected by a failure in that component. Their architecture and CAPRI share several common goals: scalability in the global Internet, the ability of diagnostic providers to control the amount of information they divulge, and reliability in the face of failure. Like the failure dependency graphs agents use in CAPRI to model the effect of component failures on other components, “cause-effect graphs” in their architecture model how failures in one part of

the network propagate to other parts in the network. The main differences between CAPRI and their architecture are that CAPRI takes a more general probabilistic approach to modeling dependencies, CAPRI provides agents with an extensible ontology that enables the diagnosis of new types of network components using new dependency knowledge, and that CAPRI does not assume that there is only one way to diagnose a failure. This is a more realistic assumption because diagnostic agents in the Internet may have different information and different methods for diagnosing failures.

Bouloutas, et al. propose a distributed system for diagnosis in which different management centers are responsible for different components [2]. The authors consider only a two phase diagnosis: first generate hypotheses and then perform tests. In CAPRI, agents may repeatedly perform tests and make inferences. Also, agents in CAPRI use a more general probabilistic Bayesian model of failures and can represent multiple levels of dependencies.

Others have developed architectures for coordinating distributed data collection and inference for certain classes of network failures. Zhang, et al. developed PlanetSeer for using both passive and active monitoring of network behavior to detect network failures [98]. A set of monitoring daemons running on hosts located throughout the Internet observe incoming and outgoing TCP packets to identify possible anomalies. When a monitoring daemon detects a suspected anomaly, it activates a probe daemon to collect data about the suspected anomaly with active probes. The results of the probe are then analyzed to verify the anomaly. PlanetSeer coordinates data collection and analysis among multiple daemons distributed throughout the network, but does not provide the facilities for describing components and dependencies as CAPRI does. Also, all the daemons in PlanetSeer follow the same diagnostic methodology and cooperate with one another, but in CAPRI I assume that each agent may employ different diagnostic methods and belong to different administrative domains.

2.1.2 Aggregation and scalable data distribution

In CAPRI, agents reduce the number of messages they must generate to diagnose multiple similar failures by reusing cached information from previous diagnoses and by selecting diagnostic actions based on their expected value and cost. Previous research in web proxy caching and multicast trees describe methods for widely distributing data to many users in a resource efficient manner [6, 7, 94]. Web proxy caches only need to match data on the basis of URLs, while agents in CAPRI must match diagnostic requests to previous diagnostic responses based on the properties of the request. There is also a great deal of research in sensor networks in efficient dissemination of information [52, 42]. The primary difference in CAPRI is that agents must perform more sophisticated reasoning to determine how to aggregate information and to whom to send diagnostic requests and responses.

2.2 Tools for diagnosis

To perform diagnosis in CAPRI, a diagnostic agent may perform diagnostic tests to determine the status of various components in the network, ranging from processes such as DNS lookup and devices such as Ethernet switches. Many researchers have developed effective

tools for diagnosing failures in various types of network components, such as Internet Protocol (IP) link failure [44], Border Gateway Protocol (BGP) misconfiguration [29], DNS configuration errors [68], network intrusion [70], performance failures [92], and so on. Each tool differs in the different types of failures it addresses, the assumptions it makes about the network, the resources it requires for diagnosis, and the speed, accuracy and cost tradeoffs that it offers. CAPRI enables the construction of agents that act as wrappers around such existing tools.

To diagnose network failures, diagnostic agents may also collect observations about the network and use this data to infer the status of network components. Recent research in network tomography demonstrate that end hosts in the Internet can collect fairly detailed information such as link-level loss rates and delay, using both passive and active measurements [90, 4]. Mahajan, et al. developed a tool called tulip that uses active probing from a single host to collect data about each link along an IP path, including round-trip loss rate, packet reordering, and queuing delay [64]. Wang and Schwartz describe a system for identifying likely network failures based on which nodes are accessible from a network management node [91]. Roscoe, et al. use a logic language called InfoSpect to detect configuration inconsistencies in a distributed system [77]. Madhyastha, et al. developed iPlane, a system for collecting and distributing measurements about various network properties such as loss rates and bandwidth [63].

Another way to diagnose failures is to determine the status of components from network measurements using statistical inference. Steinder and Sethi provide an overview of such techniques for fault diagnosis [87] and describe methods for for diagnosing network failures using Bayesian inference [86]. Ward, et al. infer the presence of performance failures based on the rate of requests processed at an HTTP proxy server and TCP connection state [92]. NetProfiler uses a peer-to-peer network to combine TCP statistics from multiple end hosts to infer reachability [67]. Shrink infers the root cause of a failure using Bayesian inference [44].

The contribution of the CAPRI architecture is to provide a common framework for communicating the wide range of diagnostic information that agents might produce using existing diagnostic tests. Such a framework enables agents of all types to share their diagnostic information and combine it to improve the accuracy, cost, and robustness of diagnosis.

2.3 Representation

An architecture for distributed fault diagnosis must provide diagnostic agents with a language for representing diagnostic information and a service description language for communicating their diagnostic capabilities.

2.3.1 Representation of diagnostic information

Diagnostic agents need a common representation for communicating diagnostic information in a way that facilitates reasoning and inference. Agents in CAPRI define component classes and properties according to a component ontology and communicate diagnos-

tic information in terms of observations, beliefs and likelihoods, and dependency knowledge. Chandrasekaran and Josephson provide a good overview on sharing problem-solving knowledge using ontologies [8]. Unlike most previous research in ontologies which focus on "domain factual knowledge" describing objective reality, the authors focus on the challenge of modeling "problem-solving knowledge", which they define as "knowledge about how to achieve various goals". According to this definition, CAPRI enables agents to represent problem-solving knowledge for fault diagnosis in the Internet. Chandrasekaran and Josephson identify the five elements of a problem-solving ontology as a problem solving goal, domain data describing the problem-instance, problem-solving state, problem-solving knowledge, and domain factual knowledge. In CAPRI, the problem solving goal is identification of the mostly likely cause of failure from a list of candidate explanations. The domain data describing the problem-instance is the set of observations, beliefs, and likelihoods that an agent has about components related to the failure. In CAPRI, the current problem-solving state can be represented in terms of the agent's *component graph* describing the properties and relationships of components in the network and a *failure dependency graph* specifying the probabilistic dependencies among component properties. Problem-solving knowledge consists of entries in an agent's *dependency knowledge base* that indicate the probabilistic dependencies among component classes. Domain factual knowledge is the knowledge to describe network components, including the core concepts of *components, properties, diagnostic information*, and the component class and property definitions in the component ontology. Chapter 4 describes component graphs, failure dependency graphs, and dependency knowledge in more detail.

A number of domain factual ontologies for communication networks exist. Quirolgico, et al. have proposed detailed ontologies for modeling networks based on the Common Information Model (CIM) [73]. This is a domain factual ontology; its purpose is to describe the components that exist in networks and their properties, not to capture the knowledge necessary for diagnosing components.

The knowledge representation approach I take in CAPRI can be thought of as a "compiled knowledge" approach as opposed to a "deep knowledge" approach [9] in that the CAPRI ontology permits agents to express diagnostic knowledge in just enough detail for diagnostic inference and does not try to capture all possible information about a failure. Rather than passing around detailed low-level "deep knowledge" about the behavior and characteristics of each component in the network, agents in CAPRI reduce this data to the essential component dependency relationships. Pieces of dependency knowledge are a form of compiled knowledge in that they reduce the full amount of detail about a component type into the minimum amount of information necessary to reason about the status of a component from the properties of other components. Chandrasekaran and Mittal argue that in general such a compiled knowledge approach can retain most of the accuracy of having deep knowledge while simplifying automated reasoning [9].

Crawford et al. describe a model for representing relationships among devices [15]. The authors argue that by modeling just a few types of device relationships such as *depends-on*, *part-of*, and a device's functional status, one can provide a general representation suitable for modeling a wide range of devices while capturing most of the essential properties necessary for diagnostic reasoning. CAPRI takes a similar approach, attempting to model component relationships generally enough to enable the description of a wide range of

network components while providing enough information for agents to reason about the causes and consequences of failures.

Other researchers have investigated the use of ontologies for reasoning about network phenomena. Pinkston et al. describe how to perform distributed diagnosis of network attacks using ontological inference [72]. The authors describe an ontology that defines different types of network attacks in terms of their characteristics, and use this information to identify various attacks based on data collected from multiple observers. The approach described by Pinkston et al. performs inference using only an OWL reasoner, but in CAPRI agents may generate new beliefs and other diagnostic information using other techniques such as probabilistic inference and data collection.

2.3.2 Representation of diagnostic services

CAPRI provides a service description language for agents to describe their diagnostic capabilities so that an agent can automatically discover and make use of the diagnostic information other agents provide. Many languages for describing network services exist today for enabling such interoperation and composition of services. OWL-S allows service providers to describe services using the OWL ontology language.[1] The Web Service Definition Language (WSDL) allows web service providers to describe SOAP web services.[2] WSDL allows a service provider to describe the allowable inputs and outputs of a service and the message exchange patterns it supports. The diagnostic service description language agents use in CAPRI provides similar functionality, enabling the description of the inputs and outputs of diagnostic services. CAPRI agents do not require the full expressiveness of OWL-S or WSDL, however, and they do not need to interoperate with other web services. In addition, CAPRI requests and responses do not require agents to maintain state across requests, which simplifies the description of the services they can provide. Therefore for conciseness and ease of implementation I choose to define a custom service description language for CAPRI.

Others have developed systems for describing, advertising, and looking up services in a network. The Service Location Protocol (SLP) allows computers and devices to advertise services within a local network [39]. Sun developed the Jini infrastructure for communicating, describing, and using distributed services in a network.[3] Unlike these generic service discovery protocols, however, the service description language in CAPRI enables the selection of actions for distributed diagnostic reasoning. The CAPRI service description language enables diagnostic agents to dynamically determine the diagnostic capabilities of other agents in terms of observations, beliefs, and dependency knowledge about network components and diagnostic tests according to a distributed component ontology.

[1] http://www.daml.org/services/owl-s/

[2] http://www.w3.org/TR/wsdl20/

[3] http://www.jini.org/

2.4 Fault diagnosis

Many approaches to fault diagnosis and related problems exist today, including root cause localization, dependency analysis, probabilistic reasoning, model-based diagnosis, case-based reasoning, fault detection, and event correlation. This section discusses related research in these areas.

2.4.1 Root cause localization

Diagnosis in CAPRI is based on the concept of root cause localization, also known as root cause analysis or fault localization. Root cause localization is the process of identifying a set of root causes that can explain a failure, and then pinpointing the precise location of the failed components. In CAPRI, a diagnostic response identifies the most likely cause of failure from a set of possible candidate explanations. In order to apply root cause localization to fault diagnosis, one must have a causal failure model. In CAPRI, this causal failure model is based on the probabilistic dependencies among properties of components and diagnostic tests. The failure of a component with dependencies may be caused by a failure in one of the components it depends upon.

Kiciman and Subramanian describe a model for root cause localization based on components and "quarks", where quarks are are the smallest observable unit of failure or success [47]. A failed quark indicates that one of the components that influence the quark has failed. For example, a quark might represent the success or failure of an AS path, while the components that influence the status of the quark are the IP hops along that path. An end user may have the ability to observe the status of the quark, but cannot directly observe the status of the IP hops. This thesis takes a similar approach and assumes that a failure observed in one component may be caused by a failure in one of its dependent components, but agents in CAPRI do not have a universal notion of quarks because each agent may be able to diagnose a different set of components. What one one agent considers to be a quark (that is, a component whose dependent components cannot be observed) may not be a quark to another agent that *can* observe its dependencies.

Steinder and Sethi describe a fault localization method in which multiple managers each perform diagnosis and then share their results [84]. The authors also describe how to model network faults using a bipartite causality graph in which the failure of individual links cause the failure of end-to-end connectivity, and then perform fault localization using a belief network [82]. Unlike the model described by Steinder and Sethi, however, agents in CAPRI can have more complex dependency models, in which end-to-end connectivity may also depend on software applications, DNS, network configuration, and so on. Like Steinder and Sethi, I model failures probabilistically because the effect of a component failure on the status of other components may be nondeterministic. Steinder and Sethi also propose incremental hypothesis updating, in which a diagnosis provider supplies a continually updated list of possible root cause faults [85].

Kompella et al. developed SCORE, a system for modeling dependencies among components in an optical IP network and performing fault localization using a minimum set cover algorithm to infer the root cause Shared Resource Link Groups (SRLG) of failures in IP paths [50]. These SRLGs correspond to the components of CAPRI. SCORE performs di-

agnosis in a centralized way, with all diagnostic reasoning occurring at the SCORE server, while CAPRI enables distributed diagnostic reasoning.

2.4.2 Dependency analysis

Root cause localization in CAPRI is based on dependency analysis. The properties of a component may depend on the properties of other components; by analyzing these dependencies, agents can determine the root cause of a failure.

Keller, et al. provide a good overview of the types of dependencies in networks and how dependency analysis can be used for root-cause localization [46]. The authors describe how dependencies may be organized along multiple dimensions according to the type of components involved, the importance of the dependency (e.g. mandatory or optional), where the components are located, and so on. In CAPRI, agents model dependencies in a more general way using probabilistic graphical models. Using such models, agents can represent many different types of dependencies, including AND dependencies where a component functions if and only if all its dependencies are functioning, OR dependencies that provide multiple alternative ways to satisfy a dependency, and probabilistic dependencies in which the status of a component can depend probabilistically on other components. Keller, et al. also describe how dependencies among software applications may be determined dynamically at runtime from software package configuration.

Grushke describes how to perform root cause analysis by searching through a dependency graph [36]. Gopal represents network dependencies according to network protocol layers [34]. Both Grushke and Gopal model dependencies among applications, network devices, and services using a directed dependency graph. Like Grushke, in CAPRI I choose to model component status using binary states (`OK` and `FAIL`). In CAPRI, however, agents use probabilistic inference to determine the status of components. In addition, agents in CAPRI can infer the status of a component from many other variables, not just the status of other components. Unlike Grushke's system and most other previous work in dependency analysis, CAPRI agents can communicate probabilistic evidence to perform inference in a distributed manner.

Sometimes an agent may not know the dependencies for a particular component individual. Fonseca et al. describe how to discover runtime dependencies for a variety of network applications by tagging messages with metadata using the X-Trace framework [32]. Bahl et al. describe a system for constructing probabilistic dependency graphs based on packet traces [1]. Gupta et al. show how to learn an accurate model of dependencies from observations of system behavior [38].

2.4.3 Probabilistic reasoning

Diagnosis requires distributed probabilistic reasoning. Reasoning must be probabilistic because many diagnostic tests only indicate a probability of failure. SCORE infers the cause of network failures using a Bayesian network in a centralized fashion [50]. Deng, et al. use Bayesian networks to infer the cause of failures in linear lightwave networks [24]. Such centralized systems are inadequate because diagnostic knowledge and data may be

distributed across multiple administrative domains. For example, a network administrator may not have the data or knowledge necessary to diagnose failures in their upstream ISP.

Crick and Pfeffer use loopy belief propagation as a way to make inferences based on sensor readings in a distributed fashion [17]. Their results suggest that communicating beliefs rather than raw sensor readings can reduce communication costs and deal with noisy sensor readings. In CAPRI, agents use a similar strategy to perform distributed diagnosis, summarizing the results of diagnostic inference using probabilistic beliefs rather than only exchanging low-level network observations.

Many researchers have studied probabilistic models for fault diagnosis [59, 17, 76]. Shrink [44] and SCORE [50] use bipartite Bayesian networks to diagnose Internet link failures. Katzela and Schwartz model dependencies among components using a probabilistic dependency graph and show how to infer the most likely cause of failure under certain assumptions [45]. Steinder and Sethi also use belief propagation to diagnose path failures [83, 88]. No common architecture for sharing diagnostic knowledge for distributed probabilistic reasoning exists, however.

Agents in CAPRI perform probabilistic inference using knowledge about dependency among classes of components. This approach is similar to previous research in methods for combining probabilistic inference with first-order logic [33, 74, 78].

Researchers in belief propagation for distributed probabilistic inference consider the cost of communicating information, but typically they only consider the cost of inference for diagnosing a single failure [71, 17, 27]. In Internet fault diagnosis, frequently multiple failures that occur over a period of time share the same root cause; in such cases CAPRI agents reduce the average cost of diagnosing each failure by caching the data obtained in previous diagnoses.

The protocol for distributed inference described in this thesis assumes discrete variables. Minka describes how to generalize belief propagation to handle inference over Gaussian beliefs [65].

Agents in CAPRI model dependencies among network components using Bayesian networks. Lerner et al. show how a dynamic Bayesian network (DBN) can describe the behavior of a dynamic system and enable the diagnosis of faults using Bayesian inference [59]. Lauber et al. use DBNs for real-time diagnosis of a mechanical device [55]. CAPRI allows agents to describe such temporal dependencies.

One challenge of fault diagnosis is choosing which test to perform next to minimize the overall cost of diagnosis. Probabilistic reasoning provides agents with a framework for selecting among multiple actions. Jensen and Liang describe several approaches to this problem, including myopic test selection, selecting tests while taking into account the impact of the information they provide on future tests, and selecting tests to balance the value and cost of tests [43]. Dittmer and Jensen present efficient algorithms for myopically computing the value of information from an influence diagram describing probabilistic dependencies among variables, the cost of performing tests, and the utility of various decisions [25]. Lee et al. describe an algorithm for selecting a minimum expected cost ordering in which to perform diagnostic tests given a component failure probabilities and dependencies [58]. Krause and Guestrin describe a procedure for choosing the optimal next diagnostic test to perform based on the cost and expected information gain of a test given a Bayesian model of components and diagnostic tests [51]. RAIL diagnoses faults in real-time by choosing

components to probe based on a global dependency model [76]. Brodie et al. consider how to minimize the cost of probing for diagnosis using a Bayesian network approach [3]. Li and Baras describe how to choose which probes to perform for fault diagnosis using belief networks [61]. Littman et al. present an approach for learning the least cost plan for testing and repairing network failures [62]. The procedure for diagnosis in CAPRI allows agents to use such cost minimization techniques to select the optimal diagnostic actions to perform.

A related problem is forming an accurate model of the probabilistic dependencies among various network components. Especially in a dynamic network in which conditions may change rapidly, an agent's model of the dependencies among components may become out-of-date and inaccurate. Kim and Valtorta describe an algorithm for detecting when a Bayesian network for diagnosis may be inaccurate and automatically constructing a more accurate model [48]. In CAPRI, a knowledge agent may use such techniques to continually provide updated dependency knowledge to other agents.

2.4.4 Reasoning and logic with incomplete information

Another approach for fault diagnosis is to make inferences from possibly incomplete information using logic and reasoning systems. Crawford and Kuipers developed Algernon, a system for knowledge representation using an access limited logic [16]. This type of logic is useful if one has a large knowledge base of facts and rules and wants to answer arbitrary queries. Since CAPRI uses a compiled knowledge approach that organizes diagnostic information and knowledge into component graphs and dependency knowledge bases, the logical inference that agents perform is relatively straightforward and does not require a system such as Algernon.

CAPRI assumes that an agent can determine the possible range of values for a component property at the time of diagnosis (e.g. a component's status is either `OK` or `FAIL`). Smyth considers how to reason about systems with unknown states [80]. Cohen, et al. describe a system for identifying the states of a system based on historical data collected about the system [14]. Dawes, et al. describe an approach for diagnosing network failures when the status of certain components may be unknown [66]. Agents in CAPRI take a more general probabilistic approach using Bayesian networks to model incomplete information.

2.4.5 Model-based diagnosis

The high-level approach that agents in CAPRI follow for diagnosing failures has some similarities to traditional model-based diagnosis. CAPRI enables agents to infer the status of components based on a model of component structure, behavior, and observed status.

Davis and Shrobe define model-based diagnosis in terms of a structural model describing the relationship between the components in a system and a behavioral model that specifies how each component behaves [19]. In CAPRI, failure dependency graphs correspond to structural models and pieces of dependency knowledge correspond to behavioral models.

De Kleer and Williams developed the GDE and Sherlock algorithms for online model-based fault diagnosis [21, 22]. De Kleer and Raiman describe a modification of the Sherlock diagnosis algorithm to take into account the computational costs of diagnosis [20].

GDE-based algorithms for fault diagnosis model dependencies among components deterministically, however, whereas the Bayesian approach used in CAPRI can also model probabilistic dependencies.

Diagnosis in the Internet differs from typical model-based diagnosis in several key respects. Typically in model-based diagnosis, the input required for diagnosis is a structural model describing the components in the system and a behavioral model describing component behavior, component status, and observations, and the output is a hypothesis about component status that explains the failure [30]. Most diagnostic systems assume no design error, assuming that the structural model accurately describes all the conditions required for correct operation. In Internet-scale diagnosis, however, an agent cannot assume that its model is complete or correct since each agent only has a limited view of the entire network. A CAPRI agent may repeatedly update its failure dependency graph based on dependency knowledge and observations it receives from other agents.

Darwiche describes a method for model-based diagnosis to efficiently compute a diagnosis based on a system structure that describes the relationships between components in terms of an acyclic AND/OR graph [18]. Darwiche assumes that the agent performing the diagnosis knows the full and correct system structure, which is not the case in Internet fault diagnosis. Katker describes a framework for modeling deterministic dependencies in a distributed system, taking into account "virtual dependencies" [54].

Other researchers have proposed methods for distributed fault diagnosis based on model-based diagnosis. Kurien et al. describe how distributed diagnosis can be viewed as a distributed constraint satisfaction problem in which diagnosers represent observations and component models using logic [53]. Debouk et al. present a method for distributed fault diagnosis in which a single coordinator combines diagnostic information from multiple diagnosers [75]. One key difference between CAPRI and these previous approaches is that agents in CAPRI can share probabilistic dependency knowledge and construct failure dependency graphs dynamically from such knowledge.

2.4.6 Case-based reasoning

Another approach to fault diagnosis is case-based reasoning (CBR), in which a case-based reasoner compares a diagnostic request to previous diagnostic requests stored in a case base and produces a diagnostic response based on similar cases. In contrast to rule-based systems that infer a diagnosis based on a set of rules, a CBR system can learn from experience, diagnose novel problems, and deal with changing conditions. Some drawbacks of CBR systems are that they require a method for retrieving similar cases and need external feedback to learn whether or not a diagnostic response satisfied the request. Lewis describes a system for diagnosing network faults based on trouble tickets using case-based reasoning [60].

Feret and Glasgow propose Experience Aided Diagnosis (EAD), which combines model-based diagnosis with case-based reasoning to help humans diagnose faults when the structural model is incomplete or inaccurate [31]. The structural model is decomposed and then model-based diagnosis is applied to each piece. Then a case-based reasoner suggests alternative diagnoses to the operator. The goal is to use the case-based reasoner to provide a human diagnoser with alternative diagnoses when an ordinary model-based approach is

inadequate. CAPRI supports similar hybrid reasoning; the diagnosis of portions of the full structural model can be done independently in a distributed fashion, and then an agent can combine this data or select a diagnosis according to a different method, such as case-based reasoning. Also, the ability of agents in CAPRI to answer diagnostic requests based on previous diagnostic information received from other agents has some similarities to case-based reasoning. Unlike Feret and Glasgow's system which is designed to assist human diagnosis, however, CAPRI is meant to be automated from the moment a request is generated until a response is returned.

2.4.7 Fault detection

This thesis focuses primarily on the problem of fault diagnosis, or determining the cause of failure once a failure has occurred. A related area of research is fault detection, or predicting when a failure will occur. Hood and Ji show how one can use a Bayesian network to model the behavior of network components to infer the presence of abnormal behavior that may lead to a failure [41]. Pinpoint detects and diagnoses failures in web services by monitoring and logging the progress of each web request [11]. Like CAPRI, Pinpoint can learn a probabilistic failure dependency model. Cohen et al. use probabilistic models to predict whether a network failure will occur based on historical metrics such as CPU usage and disk and network activity [13]. These systems only consider the detection and diagnosis of failures within a single domain, however, and do not consider the problem of distributed diagnosis of Internet failures.

2.4.8 Event correlation

Event correlation refers to the challenge of determining whether multiple alarm events are related to the same root cause. This is closely related to a problem in fault diagnosis in which a diagnostic agent wishes to determine whether a single root cause is responsible for multiple failures so it can respond to multiple diagnostic requests with the same diagnosis. Yemini et al. propose a language for specifying deterministic dependencies among components and alarms and propose a method for identifying failures based on the alarms generated [97]. Wu et al. propose a system for specifying rules for correlating alarms [95]. Liu et al. examine sequences of events to infer the possibility of a failure [37]. Klinger et al. show how to represent the effects of a failure as a code, and then matching the symptoms of a failure to a code [49]. Hasan and Sugla model causal and temporal correlations among alarms using a deterministic dependency model [40]. Chao, et al. describe a system for learning probabilistic correlations among alarms in a testbed network and use this information to diagnose failures [10], but consider only a bipartite model of network components. Unlike previous research in event correlation, agents in CAPRI describe component dependencies in a more general way using probabilistic models and perform diagnosis using probabilistic inference.

Chapter 3

Overview of Fault Diagnosis in CAPRI

The purpose of the CAPRI architecture is to enable distributed diagnosis among heterogeneous diagnostic agents in a general and extensible way while dealing with incomplete information and managing cost. This chapter more precisely defines fault diagnosis, diagnostic agents, probabilistic diagnosis, the elements of the CAPRI architecture, and the scalability challenges it addresses.

3.1 Network failures

People use the term "network failure" to refer to a wide range of phenomena, including bugs in network applications, server misconfigurations, severed links, and network congestion. Since CAPRI is a general architecture for fault diagnosis that must handle a wide range of failures, this thesis broadly defines a network failure as a perceived component malfunction. This thesis uses the term "component" not as it is commonly used in network systems to refer to physical devices, but rather in the abstract sense used in model-based diagnosis and dependency modeling. I define a component as any process or device that provides a resource or function and whose behavior can be observed and modeled. For example, a network switch is a component that provides network connectivity between the hosts that plug into it. Thus if a network administrator discovers that a switch has failed, then that is considered a network failure. Components also include processes such as TCP connections, which may fail due to the failure of other components such as network switches.

Note that I define a network failure as a *perceived* component failure. The purpose of CAPRI is not to discover and diagnose all possible network failures, but rather to only diagnose failures that users or administrators notice. For example, if a user disconnects their computer from the network, it prevents them from connecting to millions of other hosts on the Internet. Rather than attempting to detect and diagnose all of these possible failures, agents in CAPRI only attempt to diagnose failures that users actually perceive, such as a user's inability to retrieve email from a POP server. This greatly reduces the number of possible failures that agents must diagnose while ensuring that the network failures agents diagnose correspond to actual user perceived problems.

A perceived component failure is not always due to an actual component failure, however. A perceived component failure means that the user or network administrator perceiv-

ing the failure believes that the component is not behaving in the way that they expect. There are two possible reasons for this: either the component has actually malfunctioned, or the component is functioning correctly but the observer incorrectly perceived it to have failed. A diagnostic agent can model this situation by representing the true status of the component as a hidden variable while treating a failure report as a piece of evidence that can provide information about the true status of the component.

The line between functioning and malfunctioning is not always clear, however. Clearly a TCP connection with a 100% loss rate is malfunctioning, but is a 5% loss rate functional? We may choose to model the degree of failure using some scalar or vector metric, but the most appropriate metric to use to measure component performance may depend on the type of component. Furthermore, reasoning about the effects of different degrees of performance failure on other components is difficult, so this thesis only distinguishes between two possible component states, `OK` and `FAIL`. Though the distinction between functioning and malfunctioning is not always clear, the point at which agents make this distinction is not essential to this thesis. All agents must agree on the meaning of `OK` and `FAIL`, but CAPRI does not require all diagnostic agents to agree on the status of any individual component. If desired, one may construct an agent that can distinguish among more status values between `OK` and `FAIL`, but in this thesis I only consider the two state case.

This thesis focuses on fault diagnosis and not fault detection. The purpose of CAPRI is to diagnose a failure once a user has noticed it, and not to identify and report failures when they occur. There are two reasons for choosing this approach. Firstly, in a fault detection system there is no easy way for agents to determine which faults are important enough to report and which do not affect a user, so a fault detection approach may result in unnecessary processing and generate unnecessary notifications. Diagnosis only in response to user requests ensures that agents only diagnose faults that end users notice and deem severe enough to request a diagnosis. Secondly, failure detection requires an agent to constantly monitor a network component while diagnosis only needs to determine the status of a component in response to a request for diagnosis. Performing diagnosis only on request reduces the communication and processing cost of diagnosis.

CAPRI can support the diagnosis of failures in a wide range of network components, but as a starting point in my thesis I will focus primarily on the diagnosis of reachability failures for end users. This thesis focuses on "hard" network reachability failures in which a user wants to access a particular network resource such as a URI or a hostname and port number (e.g. a mail server or a web page), but is unable to do so, either because a component has failed or due to a misconfiguration. Some examples of failures include, but are not limited to:

1. Local misconfiguration. The user's network configuration is incorrect. For instance, they are using the wrong DNS servers or have their gateway incorrectly defined.

2. Link failure. The user cannot reach their ISP.

3. Routing failure. Their ISP or some other region in the Internet cannot route the user's packets to the next hop towards their destination.

4. DNS server unreachable.

5. DNS information incorrect. The DNS entry may be out of date.

6. Destination host unreachable. The destination host is down.

7. Destination port unreachable. The destination host is reachable, but the desired port is closed.

8. Destination application unavailable. The destination host and port are reachable, but the desired service is not available. For example, the HTTP server on the destination server is not responding to HTTP requests.

All of these failures can be represented as the failure of a network component and possibly some of its dependent components.

3.2 Fault diagnosis

This section defines distributed *fault diagnosis* in the CAPRI architecture. Abstractly, fault diagnosis is a process that takes as input a description of a failure from a diagnosis requester and produces as output a description of the most likely cause of failure. Unlike domain-specific diagnosis systems that assume a fixed set of possible causes of failure, however, in CAPRI the set of possible causes of failure may vary for each diagnosis and must be determined dynamically to support new diagnostic knowledge and agents with different capabilities. Keep in mind that the purpose of requesting diagnosis in the first place is to help the requester decide on a repair action. Therefore the output of diagnosis should be in terms of the possible repair actions that the requester can take. As different requesters may have different repair capabilities (e.g. a network administrator may more available repair actions than an end user), different requesters may wish to request diagnosis in terms of different sets of candidate explanations. For example, a typical user may wish to request diagnosis of web connectivity failures in terms of user network connectivity, ISP status, and destination server status; and not in terms of router configuration files and web server processes. On the other hand, network administrators or other experts should be able to request and access more detailed information. In addition, each diagnostic agent may have the ability to distinguish among a different set of possible causes. New, more sophisticated diagnostic agents may have the ability to distinguish among more possible causes of failure than previous agents. For example, one diagnostic agent might only have the ability to determine whether or not a failure was caused by a local network misconfiguration. Another agent might provide a response indicating precisely which ISP is responsible for a failure.

Therefore for extensibility to support new diagnostic agents with new diagnostic capabilities and to accommodate requesters with different requirements for diagnosis, in CAPRI the set of possible explanations for a failure is determined dynamically and may vary for each request. In CAPRI, each diagnosis provider specifies the set of explanations that it can distinguish among in its service description (see Chapter 5), and a diagnosis requester specifies which of the candidate explanations it wishes to distinguish among.

More precisely, in CAPRI a diagnosis requires as input a component perceived to have failed (e.g. an HTTP connection) and a set of candidate explanations for its failure (e.g. the

web server is down, the user's network connection has failed, the DNS lookup failed, or a failure occurred in some other part of the network). The output of diagnosis specifies the likelihood of each of the candidate explanations and identifies the most likely explanation. In addition, in order to help the requester better understand how much evidence went into the diagnosis, an agent may optionally provide additional observations and beliefs that describe the evidence used to produce the diagnosis.

Diagnosis may involve communication among multiple diagnostic agents, each of which may contribute some information useful for diagnosis. The process of diagnosis begins with an initial diagnostic request generated by a user diagnostic agent on behalf of a user. The initial diagnostic request identifies the component that was observed to have failed, supplies a set of candidate explanations for the failure, and contains a component graph describing a set of observations the user agent has about the failure. When a user diagnostic agent first begins the process of diagnosis, it may have very little information about the failure and can only provide a low accuracy diagnosis. In the process of diagnosis, agents in CAPRI gradually improve the accuracy of their diagnosis by performing local diagnostic tests and recursively requesting diagnostic information from other diagnostic agents until either its confidence in its diagnosis exceeds a certain threshold, the cost of diagnosis exceeds the allowed budget, or the time of diagnosis exceeds an expiration time. Each time an agent requests information from another agent, both the requesting and responding agents exchange diagnostic information. As diagnostic agents accumulate diagnostic information, the accuracy of their diagnosis improves.

Note that since each agent may have different methods for diagnosing failures, CAPRI supports many different patterns of diagnosis. For example, a series of agents might simply hand off a diagnostic request to the next diagnostic agent until one of them has the ability to diagnose it. Alternatively, a series of agents may each add their own observations to a failure story and pass it on to yet another agent to diagnose. Or an agent might receive a diagnostic request and then choose one of several agents to whom to pass it on to depending on the type of request. Another possibility is for an agent to first request additional data from another agent, and then forward the request on to another agent based on the data it receives. There are an endless number of possibilities, but the general pattern remains the same: when an agent receives a diagnostic request, it can repeatedly perform perform local tests or diagnosis and recursively request data and diagnosis from other agents before producing a response. Chapter 7 describes this procedure in more detail.

3.3 Diagnostic agents

Fault diagnosis in CAPRI is performed by distributed diagnostic agents. Diagnostic agents perform diagnostic tests and communicate with other agents on behalf of a user or network administrator to collect diagnostic information and perform diagnostic reasoning. The purpose of CAPRI is to support the communication of diagnostic information among heterogeneous diagnostic agents. Unlike previous systems for distributed fault diagnosis such as Planetseer [98] that assume all diagnostic agents are operated by the same administrator and know of the existence of all other agents, CAPRI agents may differ in terms of their operators, diagnosis and data collection capabilities, location in the network, technologies

they use, information and knowledge they possess, and in the way they are implemented. Diagnostic agents may be operated by ISPs, network administrators, organizations, users, or any other entity that wishes to request or provide diagnostic information. For resilience to network failures, a diagnostic agent might reside in a different part of the network than the components that it can diagnose. Such decoupling of data collection from diagnostic inference resembles the layered design of the 4D architecture for network management and control [35]. For resilience to DNS failures, diagnostic agents are identified by an IP address and port number.

Diagnostic agents may have a range of diagnostic capabilities. Agents may generate diagnostic requests on behalf of a user, produce observations of diagnostic tests, provide probabilistic information about the status and properties of various components, supply dependency knowledge to diagnose certain classes of components, or aggregate observations to produce new information. Each agent may have a different set of capabilities due to resources it possesses, its location in the network, or special technology. For example, only diagnostic agents residing within a particular AS can perform traceroutes that originate in that AS. For policy reasons, each agent may also have a different set of neighboring agents with which it may communicate. For example, an ISP might provide a diagnostic agent that only answers requests from customers of that ISP.

The heterogeneity of diagnostic agents has several implications for a protocol for diagnostic agent communication:

1. All agents must agree on a common *service description language* for describing diagnostic capabilities, including the inputs an agent requires and the outputs an agent produces so that other agents can make use of the diagnostic information it produces.

2. Multiple agents may have the capability to diagnose the same component in different ways and return different results. Therefore agents must identify the source of information in their messages to enable the resolution of conflicting messages and to identify duplicate data.

3. Agents must be able to perform recursive diagnosis. When an agent does not have the knowledge or capability to diagnose a failure, recursive diagnosis enables an agent to dispatch requests to other agents or to compose diagnostic information from multiple other agents.

4. For extensibility to support new classes of information, agents must be able to accept and communicate information about component classes and properties for which they do not have any dependency knowledge. Even if an agent does not understand a piece of data, it can pass on that data to another agent that does understand how to use it.

5. Agents should have control over how much information they reveal to other agents and are not required to reveal all information that they have. Agents may choose to reveal certain information only to authorized requesters.

Though CAPRI does not restrict the set of capabilities each agent has, it is convenient to classify diagnostic agents into five types based on their roles:

1. **User agents** that interface with users so that users can make diagnostic requests and receive diagnostic responses. User agents request additional information from regional agents to perform distributed diagnosis.

2. **Local agents** that perform diagnostic tests and make observations about a particular device or host. Frequently certain observations and tests can only be performed by a local agent residing in a particular location. A local agent may have access to information not available elsewhere, such as server logs or application error messages.

3. **Regional agents** that aggregate requests from multiple other user agents or regional agents. A single regional agent is responsible for handling diagnostic requests from a set of local agents or other regional agents. For example, an ISP might deploy a set of regional agents to diagnose failures for its customers. Regional agents have little or no specialized information on their own. Instead, they request additional information from specialist agents when necessary. As new specialist agents become available, regional agents automatically discover and take advantage of their capabilities using the procedure for diagnosis described in Chapter 7. Regional agents also act as dispatchers, deciding which agent to contact next to answer a request and where to send notifications of diagnostic information. Regional agents greatly reduce the probing and communication costs of diagnosis by aggregating multiple similar requests using the procedure described in Chapter 7.

4. **Specialist agents** that have specialized information or techniques for diagnosis. For example, a server history specialist agent might collect connection history data from users who attempt to connect to a particular server and use this information to infer the status of the server. A DNS specialist agent might be able to verify whether a particular DNS server is functioning properly or not. Specialist agents might collect data from local agents or other agents.

5. **Knowledge agents** that provide probabilistic dependency knowledge. Certain types of dependency knowledge may be well known and unchanging, such as the deterministic dependence of HTTP connection status on the status of its associated local network, DNS lookup, destination web server, and IP routing. Other types of knowledge may change more frequently and can be learned from past observations. For example, a learning knowledge agent might periodically collect information about past diagnoses from many regional and specialist agents and learn the probabilistic dependencies among properties of various classes of components. Knowledge agents provide dependency knowledge to other agents.

These divisions are not always clear-cut; each of these types of agents may have multiple diagnostic capabilities. For example, a regional agent may also be able to perform certain specialized tests. A specialist agent may act as a regional agent for a set of more specialized agents. A user agent may also act as a local agent for the user's computer.

3.4 Architectural overview

A common architecture for fault diagnosis must specify a set of representations, protocols, and procedures so that heterogeneous diagnostic agents can cooperate to diagnose failures. An architecture for distributed fault diagnosis must include the following four elements: a common, extensible representation for diagnostic information; a protocol for advertising and retrieving information about the diagnostic capabilities agents can provide; a protocol for communicating observations, beliefs, and knowledge between agents; and a procedure for agents to perform diagnostic actions for diagnosing failures in a probabilistic manner. This section provides a high-level overview of the parts of the CAPRI architecture. The next several chapters examine each of these parts in more detail.

Figure 3-1 illustrates how diagnostic agents process information. An agent obtains component class and property definitions from a distributed component ontology. Though CAPRI provides a framework for defining and communicating definitions of component classes and their properties, defining all known network component classes and properties is out of the scope of this thesis. An agent obtains service descriptions of the capabilities of other agents from a service directory. This thesis assumes a centralized directory server. For additional robustness and scalability, one may consider implementing a more distributed directory service, but the design and implementation of the directory service is not the main focus of this thesis. An agent then combines diagnostic information from other agents and local diagnostic tests into a component graph and then performs various actions to produce new diagnostic information. An agent may send this new diagnostic information to other agents or store it for future use.

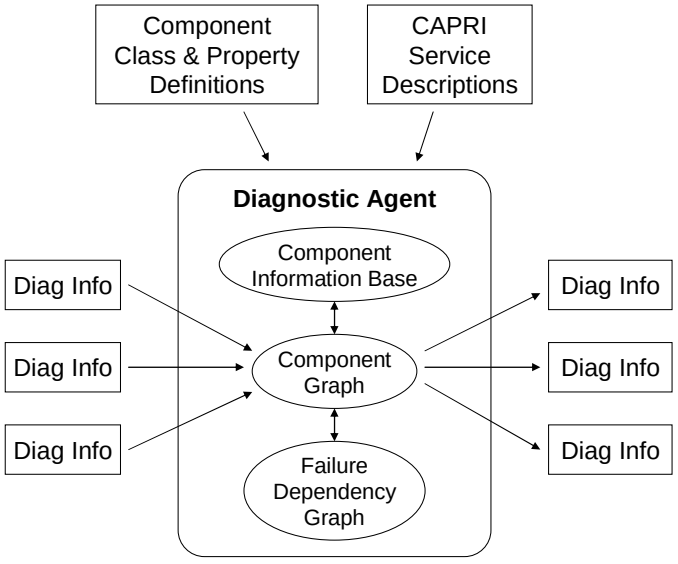

Figure 3-1: A diagnostic agent combines diagnostic information from multiple sources to produce new information using component and property class definitions and CAPRI service advertisements.

CAPRI provides diagnostic agents with a common set of representations, protocols, and procedures for fault diagnosis. These include a representation for diagnostic information,

a service description language, a protocol for communicating diagnostic information, and a dynamic procedure for processing diagnostic information.

The first element of an architecture for distributed fault diagnosis is a common, extensible representation for diagnostic information. Even though many systems for performing diagnostic tests and collecting diagnostic information exist today, no common language exists for communication of such information. A common representation for diagnostic information gives agents in different parts of the network with different capabilities and knowledge of different network components the ability to share diagnostic observations, beliefs and knowledge. Let us consider what types of diagnostic information agents might wish to communicate with other agents. Firstly, they need to be able to express *observations* and *probabilistic beliefs* about component properties, such as the results of diagnostic tests or average loss rate of a link over the past hour. They also need the ability to express *relationships* among components, such as the fact that a TCP connection traverses a particular IP link, or that an HTTP connection relied on a particular DNS lookup result. In addition, agents must agree on *component class and property definitions* in order to communicate information about component individuals and their properties. If an agent has knowledge about a new class of component or a new property of a component useful for diagnosis, it needs a way to describe that new component or property to other agents so that they can use that information for diagnosis. Also, agents may learn *probabilistic dependency knowledge* about components; they must be able to express such dependencies if they change or if new component classes and diagnostic tests become available. For example, the average probability of TCP connection failure between hosts in two different ASes in the Internet might change significantly from hour to hour as network conditions change. In addition, to help an agent decide whether to keep or discard information it receives, each piece of diagnostic information must carry *metadata* to indicate who collected the data, when it was collected, and the evidence from which the information was derived. To address all of these challenges, CAPRI defines a common language for describing network components, diagnostic tests, and diagnostic information in an extensible manner. Chapter 4 describes this language in more detail.

The second element is a language for agents to describe their diagnostic capabilities in terms of *diagnostic services* they offer. Since each agent may have different capabilities and new agents may join the system, agents need a way to describe their capabilities and understand the diagnostic capabilities of other agents. Also, each agent may require a different set of inputs in order to produce information. For example, one agent may be able to provide a belief about the probability of a web server failure given only its hostname, while another requires the web server's IP address. Therefore agents need a way to specify both the inputs and outputs of their diagnostic services so that other agents can determine which services they can use. Chapter 5 describes this service description language.

The third element of an architecture for fault diagnosis is a protocol for communicating diagnostic information while managing cost. Such a communication protocol must enable agents to request diagnostic information and respond to requests. Because every failure may involve a different set of network components, agents need a way to dynamically construct a *component graph* from information about components collected from multiple sources, and then to construct a *failure dependency graph* from this component graph to make inferences and decide what actions to perform. In order to manage both the prob-

ing and communication costs of diagnosis, a protocol for fault diagnosis must also enable agents to make tradeoffs in both the accuracy and cost of diagnosis. Chapter 6 examines these issues in more detail.

The fourth element of an architecture for fault diagnosis is a procedure for agents to select and perform actions for probabilistic diagnosis. Because an agent may receive possibly conflicting diagnostic information from multiple sources, agents in CAPRI need a procedure for managing the information they receive and using this information for diagnosis. To address this challenge, CAPRI provides agents with a procedure for constructing a *component graph* and *dependency knowledge base* to manage observations, beliefs, and knowledge received from other agents. An agent uses its dependency knowledge base to manage conditional probability tables and dependency models for classes of components that it knows and that it receives from other agents. An agent uses its component graph to keep track of observations, beliefs, and likelihoods it creates and receives from other agents, and caches information from its component graph in a *component information base* for future use to manage cost. Chapter 7 describes how agents build these data structures from the diagnostic information they exchange and then use these structures to perform distributed probabilistic inference. The procedure for diagnosis in CAPRI is general enough to support many different communication patterns and inference methods, including both centralized techniques as well as more distributed patterns such as belief propagation. In addition, I show how this procedure allows agents to manage costs using caching, evidence propagation, aggregation of requests, and confidence thresholds.

3.5 Scalability

A major strength of the CAPRI architecture is that it supports the addition of new agents, new services and new dependency knowledge. In order for agents to effectively take advantage of new services and knowledge, however, the architecture must scale as the number of agents and knowledge in the system increases. The CAPRI architecture addresses three types of scalability challenges: scalability to support many diagnostic requests, scalability to support a large number of available diagnostic services, and scalability to support a large amount of dependency knowledge.

The first type of scalability is the ability to support a large number of diagnostic requests. CAPRI addresses this challenge in several ways. First, CAPRI allows agents to cache information to reduce the cost of diagnosing many similar failures. Secondly, agents can use input restrictions to limit the requests they receive and distribute requests across multiple agents. Another technique that can reduce the cost of diagnosing multiple similar failures is evidence propagation. An agent that has evidence useful for diagnosing a failure may propagate that evidence to other agents that it believes may benefit from that evidence. Chapter 7 describes these procedures in more detail.

Second is scalability in terms of available services. As the number of services that agents can choose from increases, it becomes more costly to compute the value of all services and choose among them. Agents in CAPRI address this issue by aggregating multiple specialist services together into a more general service. CAPRI service descriptions enable agents to achieve such aggregation of services using both input restrictions and requester

restrictions. Aggregation of services reduces the number of other agents and services that each agent needs to know for diagnosis. Such aggregation resembles the way Border Gateway Protocol (BGP) routers hide the complexity of multiple routes by aggregating smaller prefixes into larger prefixes. See Chapter 5 for more details about service descriptions.

A third issue is scalability in terms of dependency knowledge and failure dependency graphs. Additional dependency knowledge and more complex failure dependency graphs can improve diagnostic accuracy, but at the cost of additional computation to perform inference. Agents in CAPRI manage such costs by decomposing dependencies into conditionally independent parts and exchanging information using belief propagation so that no single agent needs to know all dependencies. For example, an agent responsible for the diagnosis of network failures within ISP *A* does not need to know the dependencies among components within ISP *B*. Similarly, an agent that diagnoses web server status does not need dependency knowledge for diagnosing DNS servers. The hierarchical organization of the Internet simplifies the decomposition of component properties into conditionally independent parts. See Section 4.2.4 for a discussion of how agents can scalably represent dependency knowledge.

3.6 Probabilistic inference

In order to manage costs and deal with incomplete information and noisy measurements, agents in CAPRI diagnose failures using a probabilistic approach. Agents can exchange probabilistic beliefs and likelihoods about components and perform inference according to probabilistic dependency knowledge. Agents construct probabilistic failure dependency graphs and can perform distributed diagnosis according to probabilistic belief propagation.

Probabilistic inference using Bayesian networks has several advantages over deterministic dependency analysis approaches. The conditional independence assumptions of a Bayesian network facilitate distributed reasoning. For example, an agent can infer that an IP path has failed if that agent has evidence that a link along that path has failed without knowing the cause of the link failure. This structure minimizes the number of other agents with which an agent needs to communicate to infer a diagnosis. Thus each agent can maintain only a local dependency model and request additional data from a small set of other agents when required.

Probabilistic inference can greatly reduce the number of diagnostic tests required to infer the root cause of a failure compared to deterministic probing methods such as Planetseer [98]. When high-impact failures occur and an agent receives many failure requests with the same root cause, Bayesian inference enables an agent to infer the root cause with high probability without additional tests [57]. When an agent does not have enough information for diagnosis, an agent can determine which tests will provide the maximum amount of diagnostic information and perform only those tests [76].

Probabilistic inference also enables agents to provide diagnosis even when they cannot obtain accurate data due to failures or lack of information. For example, if the agent responsible for diagnosing IP connectivity failures in an Internet autonomous system (AS) *X* is unreachable, another agent can still infer the most probable explanation for a failure in AS *X* based on historical IP link failure probabilities.

Another important advantage of probabilistic inference is extensibility to take into account new dependency knowledge and evidence from new diagnostic tests. If a researcher develops a new diagnostic test that can provide evidence about a failure and provides dependency knowledge for the new test, other agents can incorporate this new knowledge and evidence to more accurately diagnose failures.

Chapter 4

Representation of Diagnostic Information

The first challenge of distributed Internet fault diagnosis I address in this thesis is the extensible representation of diagnostic information about network components and diagnostic tests. CAPRI differs from other architectures for fault diagnosis in that it provides extensibility to support new diagnostic tests and new types of information. Most previous architectures for distributed fault diagnosis only support a limited range of diagnostic tests. For example, Planetseer only supports diagnosis using traceroute results [98]. Even in more general architectures such as those proposed by Thaler and Ravishankar [89] and Bouloutas, et al. [2], agents cannot share new diagnostic knowledge about components and dependencies. The ability to share such knowledge in an extensible manner is essential due to the continually changing nature of fault diagnosis in the Internet. In the Internet, new devices and applications may appear and researchers may develop new diagnostic tests and methods for diagnostic inference. Agents must be able to communicate information about novel components or new methods for fault diagnosis that other agents do not yet understand. The challenge is to develop an extensible architecture that allows agents to take into account these changes and new information.

The choice of representation for diagnostic information in CAPRI is driven by several requirements. Firstly, it must be extensible to support the description of diagnostic information derived from new diagnostic tests about new network components. To address the challenge of extensible representation of diagnostic information, CAPRI provides agents with a component ontology language for defining new classes of components and tests and their properties to create an extensible, distributed *component ontology*. This ontology enables agents to retrieve new class definitions for classes of components and tests that they do not yet know about.

Secondly, it must support distributed diagnosis using probabilistic inference. Probabilistic inference requires the ability for agents to express both probabilistic dependencies among variables and evidence about such variables. To address this challenge, CAPRI provides agents with common representations for diagnostic information about components and tests defined in the component ontology. CAPRI agents express evidence about components and properties in terms of *observations* of *descriptive properties*. Agents express dependencies among components and properties using observations of *relationship proper-*

ties combined with *probabilistic dependency knowledge*. CAPRI separates the observation of relationship properties from probabilistic dependency knowledge to decouple the tasks of making observations and learning dependency knowledge.

Thirdly, CAPRI must support efficient diagnosis when observations of evidence and dependency knowledge are distributed among multiple agents. To address this challenge, agents can communicate probabilistic *beliefs* and *likelihoods* without revealing the evidence and dependency knowledge used to produce those beliefs and likelihoods in order to perform belief propagation for distributed inference.

Though previous researchers have developed languages and systems for describing various observations about network components [73, 93], such systems are not designed for diagnostic inference. Other researchers have developed distributed reasoning systems for making inferences using data collected from different sources [17], but do not consider how to define new classes of diagnostic information. The representation described in this chapter differs from these previous systems in that it both enables the definition of new component classes and properties as well as providing a means for agents to incorporate new information about components and diagnostic tests to diagnose failures.

4.1 Representing components and diagnostic tests

CAPRI must provide agents with the ability to represent and communicate information about network components while providing extensibility to describe new classes of network components. Network components include both physical devices such as Ethernet switches as well as abstract processes such as HTTP connections. These components may have various properties, ranging from identifying properties such as hardware MAC addresses and URIs, to component relationship properties such as the web server corresponding to a particular HTTP connection. An effective representation of network components must enable agents to express all of these types of properties so that they can form an accurate model of network components for fault diagnosis. To provide a common language for agents to describe network components, CAPRI initially provides agents with a set of a common component class and property definitions while enabling agents to define new classes of components in the future.

A component can be abstractly described in terms of the *class* it belongs to and its *properties*. A *component class* refers to an abstract category of components (e.g. web browsers), while a *component individual* refers to a particular concrete instance of a class (e.g. the Firefox web browser running on your computer). Similarly, a *diagnostic test class* is a type of diagnostic procedure that produces an observation. Examples of diagnostic test classes include *Ping Test*, *DNS Lookup Test*, and *IP Routing Test*. CAPRI treats diagnostic test classes similarly to component classes. Each component and diagnostic test individual can have *properties* associated with it. Properties of a component may include details such as the version number of a web browser; statistics such as the number of failed HTTP connections in the past hour; and relationships the component may have with other components, such as the host machine on which the browser is operating.

CAPRI provides diagnostic agents with an ontology language for defining component and diagnostic test classes and properties. Though existing ontology languages such as the

Web Ontology Language (OWL) [23] provide many useful features for defining objects and properties, agents in CAPRI only need a subset of these features to perform fault diagnosis. Some of the capabilities that OWL provides include the ability to define properties that can have more than one value and the ability to define transitive properties to allow the computation of transitive closure. Such capabilities, while conceivably useful under certain situations, are unnecessary in most cases and would greatly increase the complexity of the representation of diagnostic information and diagnostic reasoning. Therefore as a starting point, in this thesis I choose a simpler custom representation for diagnostic information. If the additional features of an ontology language such as OWL are found to be useful, they may be added in future work.

An agent in CAPRI describes diagnostic information about components using a *component graph*. Figure 4-1 illustrates a component graph describing a component individual of the *HTTP Connection* component class. Boxes indicate component individuals. Italicized text indicate component class names. Underlined text indicate identifying properties. For instance, an *HTTP Component* is identified by its destination host and connection time. An *HTTP Component* also has a number of descriptive properties depicted inside the box, including `srcIP`, `status`, and `elapsedTime`. Each of these properties may have a deterministic or probabilistic value associated with it. In addition, an *HTTP Component* has a number of relationship properties indicated as dotted lines that refer to other component individuals. This *HTTP Connection*'s relationship properties include `localNet`, `httpServer`, `dnsLookup`, and `ipRouting`. Note that a component relationship property can refer both to components whose identity is known, such as the *Local Network* component in the figure, as well as components with unknown identity, such as the *HTTP Server* component. Chapter 7 describes in more detail how agents deal with information about components whose identity is unknown. I describe the concepts of component classes and properties in more detail later in this chapter.

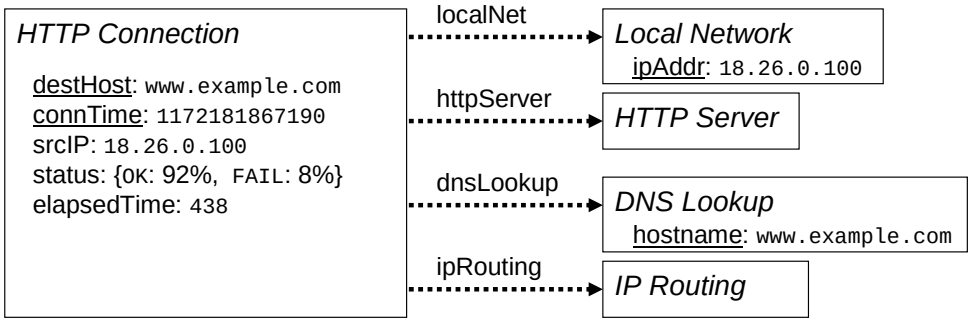

Figure 4-1: An *HTTP Connection* component identified by destination host and connection time has both descriptive properties such as `srcIP` as well as relationship properties such as `dnsLookup`.

Unlike domain-specific diagnosis systems that have built-in understanding of the meaning of various types of components and tests, for extensibility to support new diagnostic

tests CAPRI requires the explicit definition of every component class and property that agents communicate. In CAPRI, each component class, property, and diagnostic test class has a corresponding definition. To allow other agents to discover new definitions, the full name of each class and property is a URI specifying the location of its definition. For example, the full name of the *HTTP Connection* component is

http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection

One may define a new component class or property simply by placing the definition in the appropriate location. The complete class and component definition language can be found in Appendix A.1. In this thesis I may omit the full name of component classes and properties for brevity, but agents always refer to component classes and properties using the full name.

The set of all component class and property definitions comprises a *component ontology*. The component ontology provides a common language for agents to describe components and their properties to other agents, as well as define new classes of components. The component ontology can be distributed because agents in different parts of the network may contribute new component class and property definitions. In CAPRI, agents automatically retrieve class and property definitions from their associated URIs when they encounter unknown component classes and properties. This approach is based on the idea of the Semantic Web [79] in which information can be distributed across multiple locations and automatically retrieved and combined to produce new information. CAPRI provides an initial *core component ontology* at http://capri.csail.mit.edu/2006/capri/core, which defines concepts essential to fault diagnosis, such as the definition of component status. In addition, CAPRI initially provides all agents with the *common component ontology* at http://capri.csail.mit.edu/2006/capri/common, which defines a common set of well-known component classes and diagnostic tests.

The component ontology is designed to support the description of network components and diagnostic tests for the purpose of probabilistic inference. It enables agents to perform automated reasoning to combine information from multiple other agents in a distributed way. Using this ontology, agents can determine whether two components are the same or not, whether two components share a superclass, and can infer symmetric relationships.

To manage such a distributed ontology, agents need a way to keep track of the classes and properties they understand and a way to obtain definitions of new classes and properties. Each agent maintains a local table of component class and property definitions it knows about. Whenever an agent encounters a component class or property not in its class and property tables, it looks up the component or property definition using the URI corresponding to the name of the component or test class. To avoid conflicts and to distinguish between different versions of component class and property definitions, each version of a class or property definition must have a different URI. This is the same assumption made in the Semantic Web.

4.1.1 Component classes

Component and diagnostic test classes are named using URIs to provide a mechanism for agents to retrieve information about new component classes. The URI associated with a

component class or diagnostic test may or may not be hosted by a diagnostic agent. I assume that URIs used to define component and test classes do not change and are globally unique. This section describes the information provided in a component class or diagnostic test definition. In this section the term component class refers to both component and diagnostic test classes.

One challenge of communicating diagnostic information is determining whether two observations actually refer to the same component. To help address this issue, a component class definition specifies a set of *identifying properties*. Two components with the same class and the same values for each of their identifying properties are considered to be the same component. For example, suppose an agent receives observations from two different agents regarding *Ethernet Card* components. The *Ethernet Card* component class definition specifies that `MAC Address` is its identifying property. If both observations refer to *Ethernet Card* components with the same values for their `MAC Address` properties, the agent can infer that the two observations actually refer to the same *Ethernet Card* individual. Note that some classes of components may have multiple identifying properties. For example, *IP Routing* may require both a source and destination IP address for identification. If a component class does not define any identifying properties, then an agent might not be able to determine whether two components of that class are the same.

For extensibility to support new classes, CAPRI allows subclassing where each subclass may have a different set of properties and diagnostic knowledge. A component class definition for a component class *C* may specify a list of component classes *B* for which *C* is a subclass of *B*. For example, a *Wireless IP Link* class may be defined as a subclass of an *IP Link* class with additional properties such as signal strength and radio frequency. A component class inherits property definitions from all its superclasses. For example, an individual of class *Wireless IP Link* has all the properties of an *IP Link*. All component classes are subclasses of the base *Component* class. A component may be a subclass of multiple other classes. Subclass relationships are transitive. If class *C* is a subclass of class *B*, and class *B* is a subclass of class *A*, then *C* is also a subclass of class *A*. An agent can infer such relationships based on the component class definitions in the component ontology. Subclassing provides a mechanism for introducing new classes of components and tests while retaining compatibility with existing diagnostic agents.

Subclassing introduces additional challenges for agents to determine whether two observations refer to the same component. In addition to checking whether two components have the same class and the same values of each identifying property, an agent also needs to check for each combination of superclasses of each component whether they have identical values for each identifying property.

4.1.2 Component properties

Agents make use of component properties in several ways. Firstly, an agent can use dependency knowledge about one or more properties of one or more components to infer the value of one or more other properties of other components. For example, an agent might be able to infer the status of a DNS lookup if it knows the value of the IP address returned by the DNS lookup. Secondly, properties can describe relationships between components, such as the HTTP client and server of an HTTP request. Such relationship properties enable

agents to combine information about related components from multiple sources. Thirdly, an agent may use component properties for identification purposes. Like component classes, properties are identified by a URI to allow for extensibility and to retrieve new property definitions.

A property may be either a descriptive property or a relationship property. A descriptive property describes a characteristic of the component. The range of a descriptive property is a text string to simplify parsing. For example, a web server may have a descriptive property `ipAddr` that specifies its IP address. A relationship property is like a component pointer. The value of a relationship property is a component individual. For example, an *HTTP Connection* component may have a component relationship property `httpServer` whose value is the destination web server component associated with the HTTP connection. This distinction between descriptive and relationship properties in CAPRI is similar to the way in which RDF Schema[1] distinguishes between datatype properties and object properties, or how object-oriented programming languages distinguish between variables with built-in types such as integers and variables that point or refer to other objects.

An agent requires certain information about a property in order to properly make use of diagnostic information associated with that property. Firstly, an agent must be able to determine whether a property is a descriptive property or a relationship property. Additionally, for relationship properties, an agent must be able to determine the class of component to which it refers. For descriptive properties for which dependency knowledge is defined, an agent also needs to know the range of values it may take on in order to properly perform probabilistic inference. Therefore in addition to the name of the property, a property definition specifies whether it is a descriptive or relationship property, and the range of the property (a class name if it is a relationship property, or a list of values if it is a descriptive property). In addition, for defining property paths it is useful to know of any symmetric properties that may exist. For example, if component *B* is the `asPathTest` of an *AS Path A*, then *A* is the `asPath` that *B* is testing. Thus a property definition may also define such symmetric properties when they exist.

Many types of descriptive properties exist, including aggregate properties, identifying properties, and metadata properties. A property may be one or more of these types of properties. An aggregate property is a descriptive property calculated from other data and its value may constantly change. Aggregate properties include statistics about a component such as the number of bytes a web server has sent in the past hour and the average latency of an IP link. Identifying properties enable agents to determine whether two components are the same. An agent can assume that the value of an identifying property for a component never changes. A metadata property provides information about a component not inherent to the component itself. For example, one type of metadata property is `administrator`, indicating the contact information of the network administrator responsible for proper operation of that component. For the purposes of fault diagnosis, all components have a descriptive property `status`, whose value may be either `OK` or `FAIL`.

The semantics of both descriptive and relationship properties are exogenous to the system. Diagnostic agents in CAPRI do not need any domain-specific knowledge of the meaning of component classes and properties beyond probabilistic dependency knowledge, as

[1] http://www.w3.org/TR/rdf-schema/

described in Section 4.2.4. CAPRI assumes that all agents that create observations, beliefs, and knowledge about a property agree on the meaning of each class and property, though other agents that receive such observations and beliefs can perform diagnosis without understanding the meaning of all classes and properties. For example, consider a web server history agent that can observe the `consecFailuresToServer` property of an *HTTP Server* component, and a knowledge agent that learns dependency knowledge relating the `consecFailuresToServer` property of an *HTTP Server* component to its `status` property. Both the web server history agent and the knowledge agent must agree on the meaning of the `consecFailuresToServer` property for the observations and the knowledge to be consistent with one another. A regional agent that receives a `consecFailuresToServer` observation from the web server history agent and the corresponding dependency knowledge from the knowledge agent can then infer the status of the *HTTP Server* component without any knowledge of the meaning of the `consecFailuresToServer` property or the *HTTP Server* class, however.

Similarly, agents do not need to understand the meaning of relationship properties to perform diagnosis as long as all agents that create observations, beliefs, and knowledge that refer to the same relationship property agree on its meaning. CAPRI does not constrain the semantics of a relationship property. A relationship property simply points to another component, and may or may not indicate a dependency. For example, possible relationships include:

1. `verifyDNSLookupTest`: A *DNS Lookup* component may have a `verifyDNSLookupTest` relationship property whose value is a *Verify DNS Lookup Test* component testing the *DNS Lookup* component.

2. `httpConnection`: A *Firefox Error Test* component may have an `httpConnection` relationship property whose value is the *HTTP Connection* component to which the test applies.

3. `dnsLookup`: A *HTTP Connection* component may have a `dnsLookup` relationship property whose value is the *DNS Lookup* component describing the DNS lookup used to determine the IP address of the destination for the HTTP connection.

Agents in CAPRI may set the value of relationship properties using local information, the results of diagnostic tests, and domain-specific knowledge. For example, an agent might determine the value of relationship properties using a tool such as X-Trace [32] or the method described by Bahl et al. [1].

Note that component class and property definitions do not express the dependencies among components; they only specify possible properties and component relationships. The dependencies themselves are specified as dependency knowledge in the form of conditional probability tables (see Section 4.2.4). CAPRI decouples the dependency structure itself from the definition of component classes to allow the distribution of observations, beliefs, and dependency knowledge across multiple agents. For example, one agent may have observations and beliefs about a component but not have dependency knowledge for it, while another agent has dependency knowledge about that component class but not observations and beliefs. These two agents can then perform distributed fault diagnosis by communicating their diagnostic information.

Another advantage of separating relationships from dependencies is that it allows an agent to choose among multiple pieces of dependency knowledge for diagnosis. For example, an agent may diagnose TCP failures based on dependency knowledge about their source and destination IP addresses, or it may attempt to diagnose TCP failures based on dependency knowledge about the status of the underlying IP links. The approach that CAPRI agents use to represent classes, descriptive properties, relationship properties, and dependency knowledge resembles probabilistic relational models (PRMs) [33], which also decouple relationships and dependencies.

For extensibility to support new properties of existing classes, component class definitions do not specify the set of possible properties a component of that class may have, nor do component properties restrict the components that may possess them. Therefore to add a new property to an existing class, one only needs to define the new property and does not need to modify the class definition. For example, suppose Bob discovers that the number of other web pages that link to pages on a particular web server is highly correlated with the probability the web server fails. Bob might then define a new property of web servers called http://bob.example.com/capri#numLinkingWebPages and host the property definition on his own web page. He might then create a new diagnostic agent that computes this property given a web server's IP address and provides the probabilistic dependency knowledge specifying how to infer the probability of web server failure given evidence about `numLinkingWebPages`. This new specialist agent can then advertise its capabilities to other agents using a service description and provide valuable new diagnostic information for the diagnosis of web servers and other network components related to web servers. See Section 4.2.4 for more details about how an agent describes dependency knowledge and Chapter 5 for more details about how an agent can advertise new diagnostic capabilities to other agents.

4.2 Diagnostic information

CAPRI agents perform fault diagnosis using probabilistic inference. Abstractly, probabilistic inference requires both a causal model of dependencies among variables and evidence about variables from which to make inferences. In CAPRI, the causal dependency model comes from a *component graph* describing the properties and relationships among various components combined with probabilistic *dependency knowledge* describing the conditional probabilities of observing certain property values given the value of other properties. The evidence that agents use for inference consists of *observations* of the properties of network components and diagnostic test results, probabilistic *beliefs* of component property values given evidence, and probabilistic *likelihoods* of observing evidence given the value of certain component properties. Agents produce and communicate these four types of diagnostic information: observations, beliefs, likelihoods, and dependency knowledge.

I choose to decompose diagnostic information into these four types to distribute the processes of observing evidence about component properties, observing relationships among components, learning dependency knowledge, and inferring beliefs and likelihoods. This decomposition enables distributed agents in different parts of the network to perform each of the tasks of diagnosis with limited information and then combine the information to pro-

duce a diagnosis. Some agents can perform diagnostic tests to obtain observations. Others can make inferences from observations using diagnostic knowledge to form beliefs and likelihoods. Another agent can learn dependency knowledge from observations. Together, these four types of information allow an agent to infer the status of various components in the network to diagnose a failure. This section describes how agents represent each of these types of diagnostic information. Appendix A.2 provides a more detailed specification of the language agents use to express and communicate observations.

4.2.1 Metadata

To help agents decide which pieces of data to keep and which to discard, each piece of information that an agent exchanges also has some associated metadata that allows agents to determine where the data came from, when it was created, and how it was created. For example, a simple heuristic for resolving conflicting information is to prefer more recent data to older data.

Each piece of diagnostic information provides the following metadata:

1. An `originator` attribute containing the URI of the agent generating the information. This allows agents to identify the source of observations, beliefs, and knowledge in order to take into account issues such as trust and identify malicious or inaccurate agents.

2. A `time` attribute indicating the time at which the information was created represented as the number of milliseconds since Jan 1, 1970 UTC. This allows agents to determine which of two pieces of information is more up-to-date and to decide whether old data is still relevant. Note that this assumes all agents have accurate, synchronized clocks. Even if two agents have unsynchronized clocks, however, the timestamps included in the header of diagnostic messages allow an agent to compensate for large differences (see Chapter 6).

3. A Boolean `cached` attribute indicating whether the information was cached or not. If an agent receives uncached information, it can assume that the information it received is the most current information that the originator can provide. If an agent receives cached information, then it may be possible to get more up-to-date information.

4. An `expires` attribute specifying when this piece of information expires. Other agents may use this attribute to decide how long to keep cached information. Note that each piece of information may have a different lifetime.

4.2.2 Observations

Observations provide information about the properties of component individuals that agents can use for diagnosis, including the results of diagnostic tests as well as network configuration information collected by an agent. For example, an agent might make the observation that `roundTripTime` = 293 ms for an individual of class *Ping Test* with properties `srcIP` = 18.26.0.100, `destIP` = 18.26.0.1, and `pingTime` = 10:23 am. Other possible observations

include the status of a TCP connection, the hostname of its destination, or the number of times a web server has failed in the past hour. Figure 4-2 gives an example of an observation represented using XML. Appendix A.2 contains the full specification for the representation of observations and other diagnostic information.

```
<observation id="obs-1"
             time="1160577157224"
             originator="http://18.26.0.100/userAgent"
             cached="false">
  <component id="com-1">
    <class>  HTTP_Connection  </class>

    <destHost>     www.example.com  </destHost>
    <connTime>     11605771538408   </connTime>
    <status>       FAIL             </status>
    <elapsedTime>  32               </elapsedTime>
    <srcIP>        18.26.0.100      </srcIP>

    <ffoxErrorTest>  <componentRef ref="com-3" />  </ffoxErrorTest>
  </component>
</observation>
```

Figure 4-2: An observation of an *HTTP Connection* failure includes metadata about the observation, a component ID, the class of component observed, descriptive properties, and relationship properties.

In addition to the metadata attributes common to all pieces of information, each observation contains a body consisting of either a `test` or a `component` description containing a set of attribute/value pairs describing an individual test or component. In this section I use the term component to refer to both network components and diagnostic tests. Each component description in an observation contains the following:

1. An `id` string so that agents can describe the relationships among different tests and components without having to know the values of their identifying properties. It is useful to be able to identify components using a component ID as well as by the identifying properties of a component because frequently an observation may refer to a component whose identifying properties are unknown. For example, an agent may know that the status of an *HTTP Connection* depends upon the status of its associated *HTTP Server* component without knowing the identity of the web server. The component ID is unique across all components in a given component graph or message. Chapter 7 describes how agents generate component IDs and use them for identifying components received in messages from other agents.

2. The `class` of test or component represented as a URI. The class name must be a valid URI that refers to the class definition. An agent parsing this observation can use this class name to determine whether it knows about the given component or

diagnostic test class or whether it needs to retrieve this information from the ontology. Representing observation classes as URIs enables agents to extend the component and observation class ontologies with new components and observation classes.

3. A set of property names and values describing the component. These specify the actual information observed, such as the round-trip time of a ping or the destination IP address of a TCP connection. These may also specify the identifying properties of the component if available. Observed properties may include both descriptive and relationship properties.

4.2.3 Beliefs and likelihoods

CAPRI allows agents to express probabilistic evidence in terms of *beliefs* and *likelihoods*. A belief or likelihood expresses the conditional probability of an event given some evidence. A *belief* expresses the conditional probability that a property has a certain value given some evidence, and a *likelihood* expresses the conditional probability of observing certain evidence given certain property values [71]. More formally, a belief about x represents $P(x|e)$, the probability of observing each value of x given some evidence e; likelihood represents $P(e|x)$, the probability of observing the available evidence given each value of x. That is, a belief is a prediction about a variable given some evidence, whereas a likelihood expresses how the value of a variable affects the probability of the outcome observed. For example, consider an agent diagnosing an HTTP connection failure. The agent may compute a belief that the status of a web server is `OK` with 90% probability and `FAIL` with 10% probability given evidence of its historical connection status. After taking into account evidence from additional observations, however, it may determine that the likelihood of observing all the available evidence about a failure is 23% if the web server has failed and 10% if it has not failed. Therefore it produces a diagnosis stating that more likely than not, the web server has failed. Both beliefs and likelihoods are necessary to express the messages required for belief propagation. This thesis adopts the same π and λ notation for beliefs and likelihoods as Pearl [71]. Agents may infer probabilistic beliefs and likelihoods based on observations and dependency knowledge.

A belief also indicates the evidence on which it is based so that other agents can decide which beliefs to use and to avoid double-counting evidence. For example, suppose a regional agent infers the probability that an HTTP connection has failed using evidence about the number of failures to the destination web server for the HTTP connection. If the regional agent then receives a belief from a specialist agent about the status of the web server, the regional agent should only consider that new information if the new belief is based on evidence other than the number of failures to the destination web server.

There are several special cases of beliefs that are useful for distributed inference. If a belief does not take into account any evidence, then it is a prior probability distribution. If a likelihood does not consider any evidence, then the probability is 1. If a belief takes into account all evidence except the evidence downstream of a particular child i (*diagnostic evidence*), then it is $\pi_{X,i}(X)$. If a likelihood takes into account all evidence except the evidence upstream of a parent i (*causal evidence*), then it is $\lambda_{X,i}(X)$. Agents can perform belief propagation by exchanging π and λ beliefs and likelihoods.

Figure 4-3 gives an example of a belief about a *DNS Lookup* component. The belief identifies the component and property to which it applies, provides a probability distribution for the property, and lists the evidence used to derive the belief.

```
<belief id="bel-1"
        originator="http://18.26.0.100/dnsAgent"
        time="1160577157224"
        cached="true">
  <subject>
    <component>
      <class>  DNS_Lookup  </class>

      <hostname>  www.example.com  </hostname>
      <ipAddrs>  192.168.11.12,192.168.11.13  </ipAddrs>
    </component>
  </subject>

  <property>  status  </property>

  <distribution>
    <entry value="OK"   p="0.8"/>
    <entry value="FAIL" p="0.2"/>
  </distribution>

  <fromEvidence>
    <evidence propPath="verifyDNSLookupTest|dnsLookupTestResult" />
  </fromEvidence>
</belief>
```

Figure 4-3: A belief identifies the subject component and property to which it applies, provides a probability distribution table for the property, and lists the evidence used to derive the belief.

In addition to the metadata common to all pieces of information, a belief contains the following:

1. A `subject` indicating the component individual to which this belief applies. A component individual is identified either by its component class and identifying properties, or by its component ID.

2. A `property` indicating the property of the component over which the agent formed a belief. This corresponds to the variable x in the expression $P(x|e)$ or $P(e|x)$.

3. A `distribution` expressed as a conditional probability table indicating for each possible value of the specified `property`, the probability that the property has that value. For example, the status of the web server at www.example.com is `OK` with 23% probability and `FAIL` with 77% probability. Expressing probabilistic evidence

using conditional probabilities has the advantage that a conditional probability table can express any arbitrary probability distribution, but has the limitation that it can only express probability distributions for discrete variables. Minka describes how one might perform distributed probabilistic inference over continuous variables by expressing beliefs using parameterized Gaussian functions [65].

4. A `fromEvidence` field listing the evidence from which the inference was made. Each piece of evidence is described by a *property path* from the `subject` component indicating the component individual and property name of the evidence. A property path is a list of zero or more relationship properties followed by another property, and identifies a component individual and property in terms of its relationship to another component. For example, a belief about the status of a *DNS Lookup* component might be based on evidence described by the path `verifyDNSLookupTest.dnsLookupTestResult`. This property path refers to the `dnsLookupTestResult` property of the component to which the `verifyDNSLookupTest` relationship property of the *DNS Lookup* refers. This thesis represents property paths as a list of property names separated by periods ('.') or vertical bars ('|'). All properties in the path except possibly the last property must be relationship properties. Describing evidence in terms of property paths has important advantages in terms of information hiding. Firstly, it permits agents to make inferences from beliefs without needing to know the identity of the components from which the evidence is derived. Furthermore, agents can determine whether a belief provides new information without knowing the actual value of the observations that the beliefs were based on. Agents also use property paths in other contexts as well, such as to describe their diagnostic capabilities (see Chapter 5). Note that the `fromEvidence` field only specifies which properties an agent used as evidence to produce the belief, and not the values of the properties. If desired, an agent can provide the actual property values of this evidence as a separate observation.

An agent may also optionally communicate additional information about the observations and knowledge it used to produce the belief. Providing such additional information allows other agents to more effectively cache information, compute the value of available actions, and aggregate similar requests, but some agents may choose not to provide this information for policy reasons.

Agents represent likelihoods analogously to beliefs. The primary difference is that instead of a `distribution` table representing $P(x|e)$, a likelihood contains a `likelihood` table representing $P(e|x)$.

Though CAPRI does not provide agents with the ability to explicitly represent probabilistic beliefs over relationship properties, in many cases one can accomplish effectively the same thing using beliefs over the properties of a component. For example, if an agent wishes to express the belief that with 80% probability an *HTTP Connection* is to a web server with IP address 192.0.0.1 and with 20% probability the *HTTP Connection* is to a web server with IP address 192.0.0.2, it can indicate this as a belief about the value of the IP address property of the web server component.

4.2.4 Probabilistic dependency knowledge

One of the primary ways in which CAPRI differs from previous architectures for probabilistic diagnosis is that agents can exchange probabilistic dependency knowledge. Dependency knowledge describes the probabilistic relationships between component and diagnostic test properties in terms of conditional probability tables (CPTs). Unlike beliefs and likelihoods that describe probabilistic evidence about individual components and tests, probabilistic dependency knowledge applies to all individuals in a particular class. For example, if web servers in one Internet autonomous system (AS) fail more frequently than web servers in another AS, a piece of dependency knowledge about the class of all web servers might express the conditional probability a web server has failed given its AS number. Dependency knowledge may come either from human experts or from Bayesian learning. For many component classes such as IP paths, the CPTs associated with the dependency knowledge simply deterministically encode truth tables for AND or OR. In such cases application developers or other experts can easily specify the CPT. Even if the exact conditional probabilities are unknown, however, agents can learn them collectively using Bayesian learning [57].

The probabilistic dependency knowledge for a component class describes a kind of probabilistic "success story" in the sense that it allows an agent to infer the probability that a component's status is `OK` given observations and beliefs about the properties of other components in the network. Conditional probability tables provide diagnostic agents with the ability to perform both *causal* and *diagnostic inference*. Causal inference allows an agent to infer the status of a component from the status of other components it depends on. For example, if a web server has failed, then an HTTP connection to that web server will also fail. Diagnostic inference enables an agent to infer the status of a component from the status of other components that it affects or from diagnostic tests that depend on the component. For example, a failed ping test to a destination host may indicate that the host has failed.

A key feature of probabilistic inference using Bayesian networks is that it facilitates the reuse and composition of specialized dependency knowledge about related component classes. Agents can then take advantage of new dependency knowledge for inference without domain-specific knowledge. For example, someone might develop a new application class *NewApp* and create a knowledge agent that provides dependency knowledge indicating how the status of a *NewApp* component depends on the status of two related *HTTP Connections*. Suppose that another specialist agent has the ability to diagnose *HTTP Connection* failures. A regional agent that knows of the existence of both of the above agents may then offer a new diagnostic service that uses dependency knowledge about *NewApp* and the diagnostic capabilities of the *HTTP Connection* specialist to diagnose *NewApp* failures.

By collecting probabilistic dependency knowledge from other agents, an agent can gain new diagnostic capabilities and diagnose failures that it could not diagnose before. For example, an agent that learns the conditional probability of TCP connection failure given the source and destination ASes of the connection may communicate this dependency knowledge to other agents so that they can diagnose TCP connection failures as well. Or an agent might learn the probability of observing certain web browser application errors given the

status of the destination web server and the user's network connectivity. An agent may then communicate this newly learned diagnostic knowledge so that other agents can diagnose web connectivity failures given observations of web browser errors.

CAPRI enables agents to define new component and diagnostic test subclasses for extensibility while retaining compatibility with agents that do not have knowledge about the new subclasses. To support diagnosis of such new subclasses, when building a probabilistic failure dependency graph for a particular component of class *C*, an agent may use CPTs from its knowledge base for the class *C* or any superclass of *C*. For example, if an agent has the knowledge to infer the status of an *IP Link* component but does not have any knowledge pertaining to the subclass *Wireless IP Link*, then it can use its knowledge of *IP Link*s to diagnose a component of class *Wireless IP Link*. This aids extensibility by enabling agents to introduce new subclasses of existing classes without disrupting the ability of existing agents to diagnose components of previously defined classes.

Table 4.1 gives an example of probabilistic dependency knowledge expressed as a conditional probability table. Figure 4-4 demonstrates how an agent represents the knowledge in Table 4.1 using XML.

DNS Lookup Test dnsLookupResult:

dnsLookupResult	dnsLookup.status	P(dnsLookupResult| dnsLookup.status)
LOOKUP_ERROR_CONFIRMED	FAIL	0.70
LOOKUP_ERROR_UNCONFIRMED	FAIL	0.05
CORRECT	FAIL	0.01
INCORRECT	FAIL	0.10
LOOKUP_ERROR	FAIL	0.05
ALIAS	FAIL	0.09
LOOKUP_ERROR_CONFIRMED	OK	0.01
LOOKUP_ERROR_UNCONFIRMED	OK	0.02
CORRECT	OK	0.40
INCORRECT	OK	0.01
LOOKUP_ERROR	OK	0.01
ALIAS	OK	0.55

Table 4.1: An agent represents probabilistic dependency knowledge for a *DNS Lookup Test* as a table of conditional probabilities. The value of the property `dnsLookupResult` depends on the value of its parent `dnsLookup.status`.

A piece of dependency knowledge contains the following fields:

1. A `subject` indicating the component or test class for which this knowledge applies (e.g. *HTTP Connection*). The component or test class is defined in the component or test ontology.

2. A `property` indicating the property of the component for which this piece of knowledge applies (e.g. `status`).

```
<knowledge>
  <subject>  Verify_DNS_Lookup_Test  </subject>
  <property>  dnsLookupResult  </property>
  <cpt>
    <parents>  dnsLookup|status  </parents>

    <entry value="LOOKUP_ERROR_CONFIRMED"   parentVals="FAIL" p="0.70" />
    <entry value="LOOKUP_ERROR_UNCONFIRMED" parentVals="FAIL" p="0.05" />
    <entry value="CORRECT"                  parentVals="FAIL" p="0.01" />
    <entry value="INCORRECT"                parentVals="FAIL" p="0.10" />
    <entry value="LOOKUP_ERROR"             parentVals="FAIL" p="0.05" />
    <entry value="ALIAS"                    parentVals="FAIL" p="0.09" />

    <entry value="LOOKUP_ERROR_CONFIRMED"   parentVals="OK"   p="0.01" />
    <entry value="LOOKUP_ERROR_UNCONFIRMED" parentVals="OK"   p="0.02" />
    <entry value="CORRECT"                  parentVals="OK"   p="0.40" />
    <entry value="INCORRECT"                parentVals="OK"   p="0.01" />
    <entry value="LOOKUP_ERROR"             parentVals="OK"   p="0.01" />
    <entry value="ALIAS"                    parentVals="OK"   p="0.55" />
  </cpt>
</knowledge>
```

Figure 4-4: An agent represents the probabilistic dependency knowledge from Table 4.1 using XML.

3. A `parents` field supplying the list of parent variables on which the property depends. If a `parents` field is empty, the conditional probability table (CPT) for this piece of knowledge represents prior probabilities; otherwise this piece of knowledge represents conditional probabilities. Parent variables are described in terms of property paths, as defined in Section 4.2.3. Parents may include both other properties of the `subject` component as well as properties of other related components (e.g. `httpServer.status`).

4. A `cpt` (conditional probability table) containing a list of entries stating for each value and each combination of parent values for the specified `property`, the probability that the property has that value. For example, this can express the probability that an *HTTP Connection* succeeds given each possible combination of status values for the web server and the routers along the IP path to the web server. The `parents` and `cpt` fields together specify how one property (such as `status`) of a component of the specified class depends on the other properties of the component as well as properties of other components.

5. An optional `knowledgeMethod` describing the algorithm used to make create the knowledge. For example, this knowledge may have been supplied manually by an expert, or it may have been learned from observations.

A piece of diagnostic knowledge may express both prior probabilities as well as conditional probabilities. If the CPT contains only 1s and 0s, then it specifies a deterministic dependency model, such as an IP path that functions if and only if all of its constituent links function, or a DNS resolution process that functions if at least one of several alternative DNS servers functions. In general, the CPT can express any arbitrary conditional probability function of the set of parent variables. One drawback of expressing dependencies in terms of CPTs is that CPTs only support discrete variables. One way to address inference over continuous variables is to convert them to discrete variables. For example, rather than considering bandwidth as a continuous variable, one might discretize bandwidth into three values, high, medium, and low. Alternatively, to support the communication of knowledge using continuous variables (for example, latency), agents may use an alternative representation such as a parameterized Gaussian model of probabilities. Another drawback of representing dependencies as CPTs is that a CPT representation may be unnecessarily verbose for certain types of data that can be compactly represented in another form, such as a BGP routing table, for example. For this reason, two agents that wish to communicate such specialized diagnostic knowledge may choose to use a more compact alternative representation when appropriate.

Agents may also express fragments of dependency knowledge if they only know the dependencies for a fraction of the component individuals in a class. For instance, an agent might know the conditional probability of failure for TCP connections between a particular source and destination, but not for other sources and destinations. One caveat with using fragments of dependency knowledge is that if an agent does not have a complete CPT for a variable, it cannot accurately perform certain inference operations such as marginalization.

Note that it is possible to have conflicting diagnostic knowledge of a component class if the sets of parents or the probabilities do not agree. In such a case, an agent uses its local

policy to decide which CPT to use. Alternatively, an agent might attempt to produce a new combined conditional probability distribution using a noisy-OR or noisy-AND rule.

CAPRI provides an initial knowledge agent that supplies other agents with knowledge of certain dependencies among component classes in the common component ontology, such as the deterministic dependence of an *HTTP Connection* upon its associated *Local Network*, *HTTP Server*, *DNS Lookup*, and *IP Routing* components.

Chapter 5

Discovering and Advertising Diagnostic Capabilities

Each agent in CAPRI has the capability to communicate certain types of diagnostic information, described in Chapter 4. Chapter 6 presents the actual protocol that agents use for communication. This chapter describes how agents express their diagnostic capabilities and represent the diagnostic capabilities of other agents. The key challenge is to describe diagnostic capabilities in a flexible and extensible fashion to support many different types of diagnostic capabilities while enabling agents to understand and take advantage of the capabilities of other agents. For extensibility to support new diagnostic technologies, CAPRI must allow new diagnostic agents with new capabilities to share their expertise with existing agents. To address this challenge, CAPRI provides agents with a common service description language for describing diagnostic capabilities in terms of *services*. This service description language enables agents to describe the observations, beliefs, and knowledge they can provide to other agents so that other agents can take advantage of these new diagnostic capabilities. Using this language, agents can look up the capabilities of other agents and dynamically determine which agents to contact to obtain desired diagnostic information.

The strength of this service description language is it allows other agents to look up and compute the value of available services based on the inputs and outputs of a service, and not just by name. Explicitly specifying the inputs and outputs of a service provides benefits in terms of extensibility so that agents can determine the usefulness of new services without domain-specific information.

For scalability to support diagnosis using a large number of agents, this service description language also helps agents manage the complexity of selecting services in a system with many agents and services. CAPRI allows agents to specify input and requester restrictions to limit the requests that each agent receives and to reduce the number of services that each agent needs to know about.

Another advantage of dynamically selecting actions according to service descriptions is that it provides CAPRI agents with the flexibility to support a multitude of communication patterns and take advantage of new agents. Unlike previous systems for distributed fault diagnosis that only support a limited range of diagnostic communication patterns, CAPRI gives agents the ability to discover new services and the flexibility to choose among mul-

tiple alternative methods for diagnosis. For example, if an agent diagnosing an HTTP connection failure cannot contact an agent that monitors the web server's status, then it may choose to infer the probability of web server failure from user connection histories collected from other agents. The service description language described in this chapter enables agents to dynamically select an agent to contact to obtain necessary diagnostic information.

Both service descriptions and diagnostic information refer to components and properties using the same component ontology, but the service description language agents use to describe their diagnostic capabilities differs from the language that agents use to describe diagnostic information for several reasons. Firstly, the protocol for exchanging diagnostic information is different from the protocol for exchanging service descriptions; service descriptions are communicated between agents and the service directory, while diagnostic information is communicated from agent to agent. Secondly, agents use diagnostic information and service descriptions in different ways; diagnostic information is used to describe network components and dependency knowledge for inference, while service descriptions are used to describe agent capabilities and to help agents select next actions to take. Though it is possible to define a kind of upper ontology that unifies these two concepts into a single language, it is not necessary to do so in order to perform diagnostic reasoning in CAPRI. Therefore for clarity, simplicity, and ease of implementation, CAPRI defines separate languages for diagnostic information and service descriptions, though both languages refer to component and properties defined in the same component ontology.

5.1 Diagnostic capabilities and services

Before an agent can request diagnostic information, it needs to know the *diagnostic capabilities* of other diagnostic agents. Diagnostic capabilities describe the types of information that an agent can provide to requesters, including observations, beliefs and likelihoods, diagnosis, and knowledge. A diagnostic capability also specifies the input that the agent requires to produce the information it provides. Each agent may have a different set of diagnostic capabilities because each agent may have the ability to diagnose different components, perform different tests, and provide different observations. Since each agent may have different capabilities, in order to effectively perform diagnosis a diagnostic agent needs the ability to advertise its own capabilities and look up the capabilities of other agents.

An agent advertises its ability to provide diagnostic information in terms of *services*. CAPRI provides agents with a common language for describing, advertising, and looking up services. When an agent joins the system, it advertises its diagnostic capabilities in terms of service descriptions to an agent directory. Agents periodically retrieve service descriptions provided by other agents from the agent directory to dynamically discover the capabilities of other agents. This thesis assumes a centralized agent directory, but one might implement this directory in a more distributed way for additional scalability and robustness.

Each agent may advertise a different set of services depending on its diagnostic capabilities. Each agent may advertise zero or more services. Each service is associated with an individual diagnostic agent. Agents may offer four different types of services: obser-

vation services, belief services, diagnosis services, and dependency knowledge services. A service can be thought of as a remote procedure call that when given input in a certain form, produces a specified type of output. For example, an agent may provide a traceroute service that when given a destination IP address, produces observations about the routers along the IP path to that IP address.

Some agents may collect and aggregate information from other agents to produce new diagnostic information. To support such collection of information, in addition to advertising service descriptions, CAPRI allows agents to communicate *notification subscriptions*. A notification subscription requests notifications for information that matches the subscription. For example, a web server history diagnostic agent may request to receive notifications about the status of HTTP connections to a set of web servers in order to compute the probability of failure of these web servers. The concept of subscribing to notifications of diagnostic information in CAPRI is related to the idea of content-based matching in publish/subscribe systems [28]. A major challenge of content-based publish/subscribe systems is the cost of maintaining a large number of subscriptions and matching content to subscriptions. Section 5.4 describes how agents in CAPRI can address this issue by constructing aggregation-friendly agent topologies.

This thesis primarily focuses on how agents can exchange and reason about diagnostic information. Though this thesis discusses how agents can discover and make use of services other agents provide, reasoning about the relationships among services is out of scope of this thesis. Therefore as a first step, in this thesis I define services using a flat namespace. Services do not specify their relationship to other services.

5.1.1 Service advertisements

The purpose of service advertisements is to provide a requester with enough information to determine whether or not to use a service. Therefore a service advertisement specifies both optional and required inputs for the service, the output produced by the service, and the cost of using the service.

Figure 5-1 gives an example of a DNS lookup belief service advertisement. The bel:dns-lookup.status service advertised requires as input a *DNS Lookup* component and its `hostname` and `ipAddrs` properties. The advertisement restricts the set of valid requesters to regional agents. The service produces a belief about the status of the *DNS Lookup* component provided as input using the `dnsLookupResult` of a *Verify DNS Lookup Test* as evidence. In addition, this service provides an observation of the `dnsLookupResult` of a *Verify DNS Lookup Test* and dependency knowledge about the prior probabilities of the status of individuals of the *DNS Lookup* component class. Appendix A.3 contains the full specification of service descriptions.

Every service advertisement and notification subscription contains the following:

1. **Service ID**. A string to identify this service or subscription (e.g. “diag:http”). Note that a service ID has no inherent meaning and is unique only to the agent advertising the service. Another agent may advertise a service with the same service ID.

2. **Agent URI**. The agent URI and service ID string uniquely identify a service or subscription. For robustness to DNS lookup failures, agents identify themselves using a

```
<serviceAdvertisement serviceID="bel:dnslookup.status"
                      agentURI="http://18.26.0.240/dnsAgent"
                      time="1170701032"
                      messageType="beliefRequest"
                      cost="10000"
                      requesterType="regional"
                      inputClass="DNS_Lookup">

  <inputProperty propPath="hostname" required="true" />
  <inputProperty propPath="ipAddrs"  required="true" />

  <outputBelief propPath="status">
    <fromEvidence propPath="verifyDNSLookupTest|dnsLookupResult" />
  </outputBelief>

  <outputObservation propPath="verifyDNSLookupTest|dnsLookupResult" />

  <outputKnowledge subject="DNS_Lookup" property="status" />

</serviceAdvertisement>
```

Figure 5-1: A DNS lookup specialist agent advertises a belief service advertisement that requires as input a *DNS Lookup* component and its `hostname` and `ipAddrs` properties, and produces as output a belief, an observation, and dependency knowledge.

URI with an IP address (e.g. http://18.26.0.240/regionalAgent).

3. **Last modification time**. The time this service or subscription was last updated to make it easier for the agent directory and other agents to determine whether to update their service tables.

4. **Message type**. The type of service advertised. The message type may be either an observation service, belief service, knowledge service, diagnosis service, or a notification subscription.

5. **Cost**. An integer cost of using this service to help other agents decide which services to prefer. For notification subscriptions, cost represents the amount that the subscriber is willing to pay for this information. Agents might set costs based on the time this service requires to produce a response, the computational cost of a diagnostic test, or a monetary cost. This thesis assumes that costs are exogenous to the system and does not discuss how agents set costs.

6. **Requester restrictions**. An agent may choose to restrict the range of allowed requesters for a service. For example, specialist agents may choose to only answer requests from regional agents and not from user agents for scalability reasons. Similarly, a regional agent may choose to provide diagnosis only to requesters in a particular region of the network. A requester restriction may be based on credentials or on other properties of the requester such as the requester's IP address. For notification subscriptions, the requester restrictions indicate the set of requesters from which the subscriber wishes to receive information.

5.1.2 Knowledge services

In addition to the fields common to all service advertisements, a knowledge service advertisement contains the following:

1. **Output knowledge**. A list of knowledge entries this agent can provide. Each knowledge entry specifies the subject, property, and parent property paths of the knowledge (see Section 4.2.3 for the definition of property paths). If no parent property paths are provided, then the knowledge represents prior probabilities. The parent property paths are relative to the subject component of the knowledge entry. For example, an agent might advertise the ability to provide dependency knowledge for *HTTP Connection* status given `localNet.status`, `dnsLookup.status`, `ipRouting.status`, and `webServer.status`.

5.1.3 Observation and belief services

An observation service provides observations about diagnostic tests or network components. A belief service can also provide probabilistic beliefs about the properties of components. In addition to the fields contained in a knowledge service advertisement, observation and belief service advertisements also contain the following:

1. **Input class**. The class of diagnostic test or component this service requires as input (e.g. *HTTP Connection*). This is not necessarily the same as the class of component for which this service provides beliefs and observations.

2. **Input properties**. The properties that the agent requires to provide this observation (e.g. `destIP`). This is specified as a list of property paths. Input properties may refer to properties of other components as well using property path notation (e.g. `asPath.nextASHop.srcAS`). Each input property also indicates whether it is required or optional. A requesting agent must provide the value of all required properties in its request and should provide the value of all known optional properties. Not specifying an optional property may diminish the accuracy of the information produced by the service. In addition, an agent may restrict the set of valid input components by specifying an *index function* and and *index range* for a required input property. If these are specified, then the value of the index function on the input property must fall within the specified index range. Input restrictions are described in more detail below.

3. **Output observations**. A list of properties for which this service provides observations, specified as a list of property paths relative to the input component.

4. **Output beliefs**. For belief services, a list of properties for which this service provides beliefs, specified as a list of property paths relative to the input component.

5. **Output likelihoods**. For belief services, a list of properties for which this service provides likelihoods, specified as a list of property paths relative to the input component.

Note that if multiple observations match the input provided, an observation provider may supply multiple observations in the response. For example, a request for observations of HTTP connection failure to a particular destination over the past hour may return multiple results. Alternatively, an agent can subscribe to notifications to receive continuously updated information. Notifications are described in Section 5.1.5.

In many cases, an agent can only provide diagnostic information about a subset of components in a class. For example, an agent may only have the ability to perform traceroutes for *IP Routing* components whose `srcIP` is within a given range. CAPRI provides agents with the ability to specify *input restrictions* to limit the set of permissible input components that an agent accepts. An input restriction consists of an *index function* and an *index range* on a component property. An input satisfies an input restriction if the index function applied to the property value falls within the index range. CAPRI provides the following index functions:

1. `b64toint`. Converts a base64 encoded string to an integer.

2. `mod` *y*. Returns the value of $x \mod y$ for a given value of *y*.

3. `iptoint`. Convert an IP address to its network integer representation.

4. `asn`. Convert an IP address to an AS number.

An agent may compose multiple index functions in an input restriction. For example, the index function `iptoint,mod 4` first converts the property value from an IP address string to an integer, and then computes its value modulo 4.

Agents can define input restrictions for a number of purposes. One use of index functions is to distribute responsibility for a class of components among multiple agents. For example, for scalability, one might partition the set of all ISPs by their AS number and have a separate agent provide diagnostic information for each ISP. As mentioned above, another use of input restrictions is to describe the capability to diagnose only a subset of the components in a class. An agent may also use input restrictions together with costs to indicate different costs for different input components.

5.1.4 Diagnosis services

A diagnosis service returns the likelihood of several candidate explanations given a component perceived to have failed. In addition to all the parts of a belief service advertisement, a diagnosis service advertisement contains the following:

1. **Candidate explanations**. The set of properties that a requesting agent can provide in its explanations (e.g. `webServer.status`, `asPath.nextAS.status`). This allows a requesting agent to determine the level of detail that the diagnosis provider agent can provide in its diagnosis.

5.1.5 Notification subscriptions

A notification subscription resembles a service advertisement, and contains the following parts:

1. **Input class**. The class of diagnostic test or component for which to receive notifications (e.g. *HTTP Connection*).

2. **Input properties**. The properties that the agent wishes to receive, specified as a list of property paths. (e.g. `status`). This is specified as a list of property paths. Input properties may refer to properties of other components as well using property path notation (e.g. `asPath.nextASHop.srcAS`). As with service advertisements, each input property also indicates whether it is required or optional and may specify an index function and range. An agent should only send information to a subscriber that matches all the required input properties and input restrictions. Note that since input properties may refer to properties of components other than the one specified in the input class, a notification may contain information about multiple related components, and not just components of the input class.

5.1.6 Strengths and limitations

This service description language allows agents to describe a wide range of services, supporting the exchange of observations, beliefs and likelihoods, and dependency knowledge in a non–domain-specific way. It allows new agents to advertise new services that provide

new information. It enables aggregation of services for scalability using both input and requester restrictions. In addition, it allows agents to advertise services useful for tasks other than fault diagnosis, such as predicting properties of a component besides status. For example, an agent may provide a service for predicting the duration of failure for a component.

Unlike most previous systems for fault diagnosis that use a fixed procedure for diagnosis, the service description language described in this chapter allows agents to dynamically select diagnostic actions to take based on the information and knowledge an agent has available. This has several benefits. Firstly, it allows agents to take advantage of new and better diagnostic services provided by new agents. Secondly, it allows a requester agent to choose the best service for the situation based on the properties of the components in its component graph. Thirdly, it allows the modular distribution of diagnostic reasoning across multiple agents so that no single agent needs to have all the capabilities required for diagnosis. Chapter 7 describes in more detail how agents dynamically select diagnostic actions using service descriptions.

This service description language has some limitations, however. Using this service description language, an agent advertises its input restrictions and then a requester must determine whether an input component satisfies the input restrictions. Specifying input restrictions has the advantage that it allows a requester agent to select services dynamically based on the properties of components. In some cases, however, a requester agent might not know whether an input satisfies an input restriction because the agent does not know the value of the property associated with the input restriction. In such cases, a requester may either simply assume that the input is valid and ignore the input restriction, or assume that the input is invalid and not send a request. Both approaches have drawbacks, however. Assuming that the input is valid leads to unnecessary requests, while an agent that assumes the input is invalid loses the opportunity to obtain additional information.

Similarly, an agent advertising a service might not know whether or not an input component is valid until the agent processes the request. In other cases, an agent might not wish to reveal the set of inputs that it accepts for privacy or policy reasons. In both of these cases an agent may simply omit the input restriction and return an error message when the agent receives an invalid input, but doing so makes it more difficult for requesters to accurately determine the expected value of using a service.

Another limitation of this service description language is that it assumes that an agent knows what information a service will produce. In many cases, however, an agent does not know whether it can obtain the information advertised in the service until after performing a diagnostic test. One way to address this challenge is to allow the description of services that do not always return the information requested. This makes it much more difficult for a requester to compute the value of a service, however.

5.2 Service discovery

The information provided in these service advertisements enables other agents to automatically discover new services and to determine what services can provide useful information for diagnosis. Chapter 7 describes the procedure that agents use for determining what ser-

vices to use.

Each agent maintains a table of service advertisements in a *service table*. In this thesis I assume that an agent can send and receive service advertisements to and from a central agent directory server, and that all agents know the identity of the agent directory server. When an agent starts up for the first time, it advertises all the services it can provide to the agent directory, retrieves a list of available services, and stores these service advertisements in its service table. The agent directory uses the requester restriction information in the service advertisements to decide what services to provide each agent. Agents periodically poll the directory server in order to discover new services.

The combination of notification subscriptions and service advertisements enables agents to describe many different types of services, including data aggregation services. Aggregating data can be thought of as transforming one type of data into another type of data. An aggregating agent may subscribe to notifications or request diagnostic data from many other agents, and then advertise the ability to provide aggregated data. For example, an agent may offer a service that takes as input a destination web server, and provides as output the total number of recent users who have connected to it over the past hour and the number of users who could not connect.

5.3 Constructing a scalable agent topology

An important feature of CAPRI is that the service description language facilitates the creation of a scalable and aggregation-friendly topology for agent communication by enabling agents to specify requester restrictions and input property index ranges. IP routing in the Internet using BGP scales well because each router can aggregate routes on the basis of its IP prefix and route data through a relatively small number of neighboring routers. Similarly, in CAPRI, each diagnostic agent only needs to know about a relatively small number of other agents in the network. Regional agents effectively act as routers of diagnostic requests so that other agents do not need to know about all other agents in the network. User agents can send any diagnostic request to a regional agent without knowing what diagnostic capabilities other agents may be able to provide. The regional agent then decides which specialist agents to contact next. The regional agent effectively acts as a gateway router for diagnostic information, hiding knowledge and specialist agents from user agents.

Specialist agents may also act as gateways to other more specialized agents as well. For example, a generic DNS specialist agent may dispatch requests for DNS lookup beliefs to other agents that specialize in particular aspects of DNS lookups. The generic DNS specialist agent may combine information from these more specialized DNS agents and effectively hide the knowledge of the specialized DNS agents from the regional agent. An agent may dispatch incoming requests on the basis of the index ranges of input properties (for example, to achieve load balancing by distributing the responsibility of diagnosis); or based on additional information, possibly obtained from other tests. An example of the former is the way in which web server history agents are distributed by destination IP address in the prototype implementation described in Chapter 8. An example of the latter is the way in which regional agents in the prototype implementation decide whether to contact a server history agent, a DNS lookup test agent, or AS path test agent based on the

probabilistic dependency knowledge, the value of information, and the result of AS path lookups they perform.

Such hierarchical organization of diagnostic specialties has several benefits. Firstly it reduces the complexity of the service tables at each agent since an agent only needs to know about a limited number of other agents. Secondly, it makes it easier to add new agents into the system since only a relatively small number of agents need to know about a new agent. Thirdly, it promotes effective aggregation and caching of diagnostic information. By having a small set of agents handle requests for information of a particular type, those agents can effectively cache information for future requests.

One must balance several tradeoffs to construct an effective aggregation tree for diagnosis. If there are too many levels, it may lead to inefficiencies and be more prone to failure. If there are too few, information and requests may not be effectively aggregated. Similarly, if requests are distributed among too many agents, it may reduce the effectiveness of aggregation, but if there are too few agents, the agents may become overloaded. In Chapter 8 I demonstrate and evaluate several types of aggregation in a prototype implementation.

5.4 Managing the cost of notifications

CAPRI allows diagnostic agents to collect information from other agents via notification subscriptions. One challenge in CAPRI is to manage the cost of such notifications. CAPRI agents reduce the number of messages sent by batching many pieces of information together. Though batching may introduce some delay, frequently a subscriber to a certain type of information does not require the information right away. For example, in the prototype implementation described in Chapter 8, user agents batch connection history information to reduce the number of messages sent to regional agents.

One challenge of publish/subscribe systems is the cost of matching information against a potentially long list of subscribers. Unlike publish/subscribe systems for distributing news in which a large number of users may potentially subscribe to notifications of the same information, however, in distributed Internet fault diagnosis frequently only a relatively small number of agents need to know any particular piece of information. In many cases, only certain specialist agents have the ability to make inferences from a particular observation, or an observation is only useful for diagnosing failures within a particular region of the network. For example, in the prototype implementation, for any given HTTP connection observation, there exists only one server history agent to which a notification should be sent.

In those cases where many agents do wish to receive notifications of the same piece of information, frequently a data aggregation service can reduce the amount of information that must be distributed. For example, in the prototype implementation, rather than distributing connection history information to all diagnostic agents, connection history information goes only to web server history and server statistics specialist agents. These specialist agents then compute aggregate statistics such as the number of consecutive failures to a web server. Specialist agents then communicate only these aggregate statistics—or beliefs inferred using these statistics—to regional agents. The web server history agent is a type of data aggregation agent because it reduces a large amount of connection history

information into a concise aggregate statistic or belief about the status of a web server. This procedure for data aggregation both reduces the number of notifications sent and hides the details of data aggregation and belief inference from regional agents.

Chapter 6

Diagnostic Message Exchange Protocol (DMEP)

Another challenge of distributed Internet fault diagnosis is to develop a common communication protocol that enables agents to perform diagnosis using information from multiple sources, including information from new agents with new capabilities. To address this challenge, CAPRI provides agents with a Diagnostic Message Exchange Protocol (DMEP) for communicating diagnostic information, including *observations* of diagnostic test results and component properties, probabilistic *beliefs* and *likelihoods* about the values of various properties of components, and probabilistic *dependency knowledge* specifying the dependency relationships among classes of components and diagnostic tests that agents can use to make inferences from observations, beliefs, and likelihoods.

Note that unlike previous architectures for communicating diagnostic information such as Sophia [93] that only allow communication of observations of component properties, or belief propagation algorithms that only enable the exchange of probabilistic beliefs [17], CAPRI also enables agents to communicate probabilistic dependency knowledge about classes of components. Exchanging such dependency knowledge gives agents the ability to learn and accumulate new dependency knowledge to take into account new diagnostic technologies and changing network conditions. For example, an agent may learn that with 90% probability, when three different users cannot connect to a web server in the past minute, the web server has failed. An agent may then communicate this dependency knowledge to other agents who can then diagnose web server failures with greater accuracy and fewer active probes. DMEP also provides agents with the information necessary to make effective accuracy/cost tradeoffs and resolve conflicting information.

Recall that in addition to the communication of diagnostic messages, agents also communicate service descriptions (see Chapter 5) and may retrieve component class and property definitions (see Chapter 4). This chapter discusses only the communication of diagnostic messages.

6.1 Communicating diagnostic messages

CAPRI agents communicate diagnostic observations, beliefs and likelihoods, dependency knowledge, and diagnostic results with one another by sending messages. A single message may contain multiple pieces of diagnostic information. Diagnostic agents may create new information as they process messages they receive from other agents. While processing a message, an agent may send additional requests and notifications to other agents.

A message from one agent to another is either a request, a response to a request, or an asynchronous notification. Requests and responses allow agents to request particular pieces of information from other agents, while notifications allow agents to communicate new information as it becomes available. The types of messages correspond to the types of services described in Chapter 5. Each type of request has a corresponding type of response.

1. **Observation request**. The requesting agent requests an observation of a property of a component from a provider agent. The provider agent may already have this information or it may need to perform a diagnostic test to collect it. Responding to observation requests does not require any probabilistic inference. Examples of observation requests are requests for connection histories, requests for the results of a diagnostic test, and requests for aggregate statistics about a component.

2. **Belief request**. A belief request produces a probabilistic belief about the value of a component property. Belief requests include requests for the probability that a particular component has failed, or the likelihood that one of a number of other components has failed given a known component failure.

3. **Knowledge request**. The requesting agent requests a piece of dependency knowledge from another agent for a property of a component of a particular class. For example, an agent may wish to know the probabilistic dependencies for the status of a class of component or the prior probability of a property for components of a particular class.

4. **Diagnosis request**. The requesting agent requests diagnosis from another agent. The diagnosis provider agent may do additional tests and recursive requests to diagnose the failure. In a request for remote diagnosis, the requester wishes to know the likelihood of each of a set of possible explanations for a failure. Unlike a belief request, a diagnosis response indicates the probability that a set of properties have particular values whereas a belief response only provides the probability distribution for a single variable.

5. **Notification**. Asynchronous notifications allow agents to communicate new information when it becomes available, including new observations, beliefs, and knowledge.

6.1.1 Protocol requirements

To support both diagnosis on demand as well as the asynchronous propagation of diagnostic information, DMEP allows the communication of both requests and responses as well as

asynchronous notifications. In designing a protocol for distributed diagnosis in the Internet, we must consider the cost of diagnosis. DMEP provides several mechanisms for agents to control the cost of diagnosis.

In order to prevent loops and runaway explosions of diagnostic messages, each diagnosis request contains a budget, confidence threshold, and expiration time to bound the number of messages produced in response to a diagnosis. A cost budget or a confidence threshold for diagnosis allow agents to trade off accuracy and cost. Agents also specify an expiration time to bound the amount of time a diagnosis can take; sometimes a rapid diagnostic response is more important than a slow but accurate response. These mechanisms for limiting the number of messages generated are similar to the idea of decrementing a TTL when routing an IP packet. Another technique for preventing certain loops is to not use a service if its service description indicates that it does not provide any new information. In addition, using the procedure for diagnosis described in Chapter 7, for any given diagnosis request, a diagnostic agent will not reuse the same service with the same input. This prevents certain infinite loops by ensuring that every time an agent performs an action, either the set of available actions decreases or the agent gains some additional information. Note that simply recording the diagnostic agents who have contributed to diagnosis is not enough to prevent looping because an agent may legitimately contribute multiple times to a diagnosis due to the incremental, iterative nature of diagnosis.

6.1.2 Message format

Each message has the following format:

1. Header indicating the type of message: observation request, belief request, knowledge request, diagnostic request, notification, observation response, belief response, knowledge response, or diagnostic response.

2. The `requesterID` of the requesting agent as a URI written using the IP address of the agent.

3. The `serviceID` indicating the service that the agent is requesting. The service ID must match the service ID provided in the service description of the responding agent.

4. Message-type specific headers.

5. An message `body` containing additional diagnostic data, beliefs, and knowledge. For example, if a diagnostic agent wishes to pass on all the data it has about a failure for another agent to diagnose, this body is a failure story comprising all the observations and beliefs it has related to the failure.

Below I describe in more detail the format of each type of diagnostic message. Please refer to Appendix A.2 for examples of diagnostic messages and the detailed specification of the protocol.

6.1.3 Observation and belief requests

Observation and belief requests have similar forms. An observation or belief request contains the following message-type specific headers:

1. A `requestID` string.
2. An integer `expires` indicating the expiration time of the request specified as the number of milliseconds since Jan 1, 1970 UTC. If the expiration time passes, then the agent does not perform any more diagnostic actions and returns the information it has collected.
3. An integer `budget`. Each diagnostic test or inference involves some computation cost or network resource usage. The cost of diagnostic tests and inference is deducted from the budget every time an agent performs a diagnostic action. If the budget reaches zero, the request is dropped. To allow for further refinements later, the default cost of a diagnostic action is 10. To bound the number of diagnostic messages generated by a request, the default budget is 1000.
4. An `inputComponent` that satisfies the input requirements of the service description for the observation or belief request.

The body of an observation or belief request must provide the required inputs specified in the provider agent's service description for the requested service. Note that even though an observation request provides only a single input component, a requester can supply observations and beliefs about other components in the body of the message. For example, an observation request may provide an *HTTP Connection* component as input, but also provide information about the hostname of the *DNS Lookup* component associated with the *HTTP Connection*. The requester can indicate the relationship between these two components using relationship properties.

Belief requests have the same format as observation requests. A request for belief asks another agent to compute the probability some property of a component has a particular value x given any evidence e that the agent has. A belief represents the value of $P(x|e)$.

6.1.4 Knowledge requests

In addition to the header fields common to all requests, a knowledge request contains an integer request ID. No additional information is required in a knowledge request.

6.1.5 Diagnosis requests

A diagnosis request asks for the likelihood of several candidate explanations for a component failure. A diagnosis request specifies a list of candidate explanations, identifies the component that has failed, and may provide the value of some of the component's properties and the value of the properties of other related components as well. When requesting remote diagnosis, an agent may optionally provide additional observations and beliefs (i.e. the failure story) in the body of the request. The diagnosing agent may then incorporate

the evidence in this failure story in its diagnosis. Additionally the output may include additional observations, beliefs, and knowledge not directly related to the component under diagnosis in order to assist in future diagnoses.

The primary difference between a diagnosis request and a belief request is that a diagnosis response indicates the probability that a set of variables has a particular value whereas a belief response only provides the probability distribution for a single variable. This allows an agent to answer questions such as, "Is my HTTP connection failure caused by the failure of the destination web server or the failure of my network connection to my ISP?" In addition, a diagnosis provider usually performs additional tests and requests to achieve the desired confidence level.

Diagnostic requests include all the headers of a belief request with the addition of the following:

1. A set of candidate explanations, where each explanation specifies a set of components and their statuses. These explanations come from the candidate explanations provided in the service description.

2. Confidence threshold. If the confidence in a diagnosis exceeds this threshold, then no further diagnostic tests or inference need be performed. Each user requesting diagnosis in the Internet may have different accuracy and cost requirements for diagnosis. Some may want more accurate and precise diagnosis while others may only want to know the approximate location of failure. A confidence threshold enables requesters to choose between a fast and inexpensive diagnosis versus a more costly and more accurate diagnosis.

A diagnostic request may contain additional observations about the perceived failure (the "failure story") in the body of the message. Observations may include error codes from an application, recent events, configuration files, run-time dependency analysis, and so on. The service description for a diagnosis service specifies a list of both required and optional input properties that are useful for diagnosis. A diagnosis requester can use this service description to decide what information to include in a diagnostic request. For example, the input properties of a service description for an HTTP connection failure diagnosis service may include the source IP, destination hostname, the time the failure occurred, the number of recent failures the user has experienced, the application error message, and whether the user is using a web proxy. The user agent can provide observations of all these properties in the body of a diagnostic request. Note that an agent may include observations in its failure story that it does not know how to use for diagnosis because such observations may be useful to other agents.

6.1.6 Notification

A notification specifies the service ID of its associated notification subscription. The body of a notification contains a list of one or more observations or beliefs. The body must contain all the required input properties specified in the notification subscription.

6.2 Responses

When an agent receives a request, it first needs to determine whether it can answer the request. If the time has already expired, it does not have the capability to respond, the requester does not have permission to make the request, or if the budget is insufficient, then it may refuse the request and return an error message.

Each response includes the integer request ID of the request for which it is a response. The body of the response contains the information requested by the agent. A response body may also include additional useful information that was not explicitly requested.

6.2.1 Observation response

To answer an observation request, an agent returns the requested information either from its cache or by conducting a diagnostic test using the method indicated in the service description. More generally, the output of a test may include multiple observations.

6.2.2 Belief response

When an agent receives a belief request, it may return its existing beliefs or it may choose to perform additional tests to improve the accuracy of its beliefs. A belief response has almost exactly the same form as an observation response, except that it may contain beliefs as well as observations.

6.2.3 Diagnostic response

A diagnostic response reports the likelihood of each candidate explanation provided in the request. When an agent receives a diagnostic request, it may choose among multiple methods for performing diagnosis. Chapter 7 describes the process of diagnosis in more detail. Each time an agent requests information from another agent, it deducts the cost of the the operation from its budget.

A diagnostic response may include all the types of data in a belief response. The response should contain at least the amount of detail specified in the agent's advertisement. A response may also include additional observations, beliefs, and knowledge to justify the explanations provided. A diagnostic response must contain the likelihood of each of the candidate explanations specified in the original diagnostic request. The confidence in the most likely explanation is defined as the likelihood of the most probable explanation given the evidence that the diagnosis provider has.

6.2.4 Knowledge response

A knowledge response provides the requested knowledge.

6.2.5 Error messages

There are several types of errors that can occur. Some types of error messages include the following.

1. Insufficient input. An agent needs additional inputs to perform a service. This error message specifies what additional data is necessary.

2. Permission denied. An agent may refuse a request from a requester with insufficient permissions.

3. Request refused. An agent may decline a request for now, but accept requests in the future.

4. Insufficient budget. The cost of responding to this request is greater than the supplied budget.

5. Request expired. The expiration time has passed.

6.3 Messaging protocol

Many possibilities exist for the choice of transport protocol. In order to simplify adoption and implementation, I choose an approach based on HTTP and XML.

The messaging protocol in CAPRI resembles existing remote procedure call protocols such as XML-RPC[1] and SOAP.[2] XML-RPC does not easily support the complex data structures that diagnostic agents exchange, however, and the complexity of SOAP lessens its appeal for use as a messaging protocol. Therefore I decide to implement messaging in CAPRI using a custom protocol based on HTTP.

To circumvent firewalls and network address translation (NAT) boxes and for ease of implementation I choose to implement DMEP over HTTP. Every diagnostic agent is identified and accessed using a URI. A diagnostic agent might run as a dedicated web server, or a single web server might multiplex requests for multiple diagnostic agents having different URIs on the same host and port number. Asynchronous notifications are implemented as HTTP POST requests. A DMEP request is sent as an HTTP POST containing the request message using the responding agent's URI as the resource name. The HTTP response contains the content of the DMEP response. For resilience to DNS failures, diagnostic agents advertise their URIs in terms of IP addresses rather than DNS names. Note that a response may require a long time to complete because a diagnostic agent may perform many diagnostic tests and contact other agents to produce a response. In case the connection is reset or interrupted, an responding agent may reply to the original requesting agent by sending the response as an HTTP POST message to the requesting agent's URI.

Another advantage of using HTTP as a transport protocol is that it facilitates the implementation of diagnostic agents using existing HTTP server software. In addition, client

[1] http://www.xmlrpc.com/

[2] http://www.w3.org/TR/soap/

support for making HTTP requests is widespread. Furthermore, HTTP supports authentication and encryption, which is important for security.

HTTP has some drawbacks, however, such as verbose headers and awkwardness of implementing server-initiated messages (See http://www.ietf.org/internet-drafts/draft-ietf-netconf-soap-08.txt). Nevertheless these drawbacks are not fatal and are offset by the practical advantages of easier adoption and implementation of HTTP as a transport protocol. Other possibilities for a messaging protocol include BEEP, SMTP, and constructing a custom protocol.

With HTTP as the messaging protocol, each response should have an HTTP return code of 200 OK and content-type of text/xml. A request is a POST with content-type text/xml. For maximum ability to get around firewalls and NAT boxes, the recommended port number is port 80, the default for HTTP. If it is not possible to use port 80 because another application is using it, then port 8111 is recommended.

In addition, to reduce the number of HTTP requests, an agent may batch together multiple messages in a single HTTP request. Agents may also make parallel requests to multiple agents simultaneously.

The contents of the messages themselves is XML because of widespread support for parsing and serializing data to and from XML. The advantages of XML over a plain text or custom format include the ability to support Unicode and the ability to easily extend the DMEP protocol in the future. Though other formats such as N3[3] exist for concisely describing information, parsers for such languages are not as widely available as for XML. The main disadvantages are the additional parsing and serialization time, increased message size, and verbosity.

[3]http://www.w3.org/2000/10/swap/Primer

Chapter 7

Message Processing Procedure

The previous chapters described how CAPRI agents can represent and communicate diagnostic information with one another in terms of diagnostic messages. Simply being able to communicate diagnostic information is insufficient for diagnosis, however; agents also need a common procedure for making sense of the information they receive and deciding what actions to take next based on this information. For extensibility, this procedure must allow agents to process new information without domain-specific knowledge. For scalability, this procedure must facilitate aggregation of similar requests to reduce the cost of diagnosing a large number of failures. CAPRI addresses these challenges by providing agents with a general and dynamic procedure for agents to perform diagnosis while coping with communication costs and network failures.

The strength of this message processing procedure is that it gives agents the ability to compute the value of new services and to incorporate new observations, beliefs, likelihoods, and dependency knowledge from multiple other agents without domain-specific knowledge. Using this procedure, agents can dynamically choose which agents to contact and what actions to take next. Most previous systems for distributed fault diagnosis support only static communication patterns and cannot take advantage of new capabilities offered by new agents. Some systems collect data in a distributed way and then perform diagnostic inference at a single centralized location [98, 50]. Others potentially require all diagnostic agents to participate in diagnostic inference [17, 27]. In a real network fault diagnosis scenario, however, network failures, cost considerations, and the heterogeneous capabilities of diagnostic agents may restrict communication among diagnostic agents. For example, a network failure might prevent an agent from obtaining a particular piece of diagnostic information, forcing it to choose an alternate strategy for diagnostic communication, data collection, and inference involving a different set of agents and information.

Thus agents need the ability to dynamically choose among multiple communication patterns for fault diagnosis. For example, a series of agents might simply hand off a diagnostic request to the next diagnostic agent until one of them has the ability to diagnose it. Alternatively, an agent may first request additional data from another agent, and then forward the request on to another agent based on the data it receives. Many other possibilities exist as well. The most appropriate agent to contact next depends on many factors, including the capabilities of available agents, the observations and beliefs collected about a failure, and the probabilistic dependencies among components. CAPRI provides agents

with a flexible procedure for fault diagnosis in which agents can process the information they receive from other agents to form a *component graph* representing its beliefs and observations about the network and apply probabilistic dependency knowledge to construct a *failure dependency graph* for performing fault diagnosis. Agents can then dynamically compute the value of available diagnostic services and select an action to perform while taking into account both the accuracy and cost of diagnosis.

Another major challenge of fault diagnosis in the Internet is managing cost. Costs include both the cost of performing diagnostic tests (probing cost) and the communication cost of exchanging diagnostic information. Previous research in minimizing the cost of diagnosis considers only the cost of making diagnostic tests and not the cost of communication among multiple diagnostic agents in the Internet [76, 58, 51]. Communication costs can be substantial, however. The large number of hosts in the Internet creates the potential for concentration of the costs of diagnostic probes and diagnostic messages, leading to implosion. Consider a serious failure that simultaneously affects millions of users in multiple administrative domains (e.g. Internet autonomous systems). If all affected users simultaneously request diagnosis from a single agent, the implosion of requests can overwhelm the diagnostic agent. If instead each affected user attempts to conduct their own diagnostic tests to diagnose the failure and ping the failed destination, their tests may trigger intrusion detection systems and cause denial of service.

Also, diagnosis occurs repeatedly, especially when a serious failure affects multiple users that notice the failure and request diagnosis over a period of time. Previous researchers have considered how to minimize the cost of probing for a single act of fault diagnosis [61, 76, 51], but since evidence gathered in one time period may provide information for future diagnosis, minimizing the cost of multiple diagnoses over a period of time may require a different pattern of diagnosis than minimizing the cost of each individual diagnosis. Therefore agents can greatly reduce the cost of diagnosis for serious network failures by aggregating similar requests. This chapter describes how agents in CAPRI can control both the probing and communication costs of diagnosis, preventing implosion and trading off accuracy and cost using confidence thresholds, evidence propagation, caching, and aggregation trees.

Yet another challenge of fault diagnosis in the Internet is dealing with incomplete or probabilistic information and diagnoses. Due to network failures, differing diagnostic capabilities, and the cost of diagnostic tests and communication, agents must be able to diagnose failures with as little or as much evidence as they have available. Many diagnostic tests are imperfect and can only give probabilistic evidence of failures. To address the challenges of incomplete and probabilistic information, CAPRI provides a procedure for agents to dynamically combine observations, beliefs, and dependency knowledge to probabilistically diagnose failures with incomplete information without domain-specific knowledge.

The message processing procedure described in this chapter addresses the challenges listed above using several techniques. Firstly, to deal with incomplete information and distributed dependency knowledge, an agent diagnoses failures using distributed probabilistic inference. Secondly, to incorporate new diagnostic information without domain-specific knowledge, an agent combines information from multiple sources to construct a *component graph*. Thirdly, an agent reduces the cost of diagnosing multiple failures by caching component information in a *component information base*. Fourthly, in order to diagnose

failures using new evidence and new dependency knowledge, an agent dynamically constructs a *failure dependency graph* from a component graph and a *dependency knowledge base*. Finally, to take advantage of new services provided by new agents, an agent dynamically computes the set of possible next actions to take and the value of each action using a failure dependency graph.

Together, these capabilities provide CAPRI agents with a general procedure to diagnose failures in a distributed manner without domain-specific knowledge. This procedure is general enough to support many different communication patterns and inference methods, including both centralized techniques as well as more distributed patterns such as belief propagation. In addition, I show how this procedure allows agents to manage costs using caching, evidence propagation, aggregation of requests, and confidence thresholds.

Agents follow the same general procedure for processing all types of incoming messages. At a high level, the procedure is as follows:

1. Parse an incoming message to construct a component graph.
2. Retrieve component class and property definitions for any unknown classes and properties.
3. If the incoming message is an observation or belief request:
   (a) Perform the requested observations and infer the requested beliefs.
4. If the incoming message is a diagnostic request, repeat until done:
   (a) Incorporate cached diagnostic information into the component graph from a component information base.
   (b) Construct a failure dependency graph from the component graph.
   (c) Infer the cause of failure.
   (d) If the agent's confidence in its diagnosis is sufficient, the budget has expired, or the expiration time has passed, return a diagnosis.
   (e) Compute the value of each possible diagnostic action.
   (f) If no actions are available, return a diagnosis.
   (g) Perform the diagnostic action with greatest utility.
   (h) Incorporate any new information into the component graph and dependency knowledge base.
5. Save the component graph to the component information base.
6. Return a response containing the requested information.
7. Send notifications to the appropriate subscribers if necessary.

This chapter discusses each of these steps in more detail and describes the data structures and algorithms involved in message processing. First I describe diagnosis using probabilistic inference and the output of diagnosis. Next I discuss the data structures involved in message processing. Then I explain how agents can select diagnostic actions and incorporate information from multiple sources for inference.

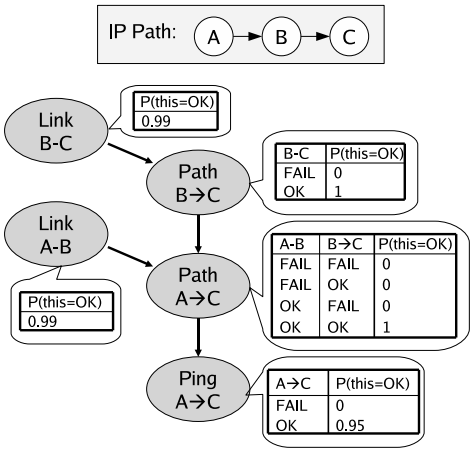

Figure 7-1: A Bayesian network for IP path diagnosis allows agents to infer the status of a component from available evidence.

7.1 Probabilistic inference

To address issues of incomplete information and distributed dependency knowledge, diagnosis in CAPRI is based on the principle of probabilistic inference using Bayesian networks. Diagnosis using probabilistic inference has several important advantages over deterministic rule-based approaches. Firstly, Bayesian networks can model both deterministic and probabilistic dependencies among many types of Internet components and diagnostic tests. Probabilistic dependencies may model both causal knowledge of the variables that influence a component property as well as diagnostic knowledge of tests that reveal evidence about a component property. For example, a piece of causal knowledge may specify that an IP path functions if and only if the first hop link functions and the rest of the path functions. A piece of diagnostic knowledge may state that a ping along that path will always fail if the path has failed, but may fail 5% of the time even when the path is functioning. Individual links function 99% of the time. Figure 7-1 illustrates a Bayesian network for diagnosing the path $A \rightarrow B \rightarrow C$. Using this network, an agent can infer, for example, the conditional probability that Link *B*-*C* has failed given evidence that Ping $A \rightarrow C$ has failed and Link *A*-*B* is functioning. An agent's failure dependency graph represents dependencies and evidence in terms of such a Bayesian network. To take into account evidence from active probing or changing network conditions, an agent rebuilds its failure dependency graph whenever its component graph or dependency knowledge base changes.

The conditional independence assumptions of a Bayesian network facilitate distributed reasoning and enable the distribution of dependency knowledge and diagnostic capabilities among multiple agents. For example, an agent can infer that an IP path has failed if that agent has evidence that a link along that path has failed without knowing the cause of the link failure. This structure minimizes the number of other agents with which an agent needs to communicate to infer a diagnosis. Thus each agent can maintain only a local dependency model and request additional data from a small set of other agents when required.

Probabilistic inference can greatly reduce the number of diagnostic tests required to infer the root cause of a failure compared to deterministic probing methods such as Plan-

etseer [98]. When high-impact failures occur and an agent receives many failure requests with the same root cause, Bayesian inference enables an agent to infer the root cause with high probability without additional tests [57]. When an agent does not have enough information for diagnosis, an agent can determine which tests will provide the maximum amount of diagnostic information and perform only those tests [76].

Probabilistic inference also enables agents to provide diagnosis even when they cannot obtain accurate data due to failures or lack of information. This is a crucial advantage because frequently diagnostic agents do not have access to all information about a failure. For example, if the agent responsible for diagnosing IP connectivity failures in an Internet autonomous system (AS) X is unreachable, another agent can still infer the most probable explanation for a failure in AS X based on historical IP link failure probabilities.

Bayesian inference has some limitations, however. Firstly, if the dependency structure does not exhibit much conditional independence, then it may not be possible to modularly decompose the dependency knowledge for diagnosis. Secondly, if the model assumes conditional independence when variables are not actually conditionally independent, then inference may produce an incorrect result. In practice, however, such problems may not be very serious. Although assumptions about conditional independence may not always be correct, in practice agents can still use probabilistic inference to produce reasonably accurate results most of the time.

Unlike domains such as medical diagnosis, typically the consequences of an incorrect diagnosis in Internet fault diagnosis are not severe. If a diagnostic agent can quickly and accurately diagnose the majority of failures, it can greatly assist a human network administrator. For the more difficult to diagnose failures and in the cases where the diagnosis is incorrect, a human administrator can manually perform additional tests and inference as they do today.

7.1.1 Output of diagnosis

In CAPRI, the task of fault diagnosis is to decide which of a number of candidate explanations for the failure is most likely. A diagnosis provider specifies the set of candidate explanations that it understands, and the diagnosis requester indicates which of the explanations they wish to distinguish among. The appropriate set of candidate explanations to consider depends on the level and type of detail that a diagnosis requester desires. For example, a typical user when faced with a failure to connect to web server may want to know who is responsible for the failure: themselves, the web server, their ISP, or someone else; whereas a network administrator may wish to know exactly which component has failed. Having a list of candidate explanations to consider enables an agent to provide the appropriate amount of detail in the diagnosis. To diagnose a failure an agent collects evidence to identify which explanation is most likely. A candidate explanation h is simply a set of one or more variable assignments. Then given a set of n candidate explanations $H = \{h_1, \ldots, h_n\}$, the task of a diagnostic agent is to collect evidence e to identify the most likely candidate explanation h^* that explains the evidence e:

$$h^* = \underset{h_i \in H}{\operatorname{argmax}} P(h_i|e) \tag{7.1}$$

An agent also returns the value of $P(h|e)$ for each candidate explanation h to the requester.

Note that the common approach of diagnosis by computing the most probable explanation (MPE) of the available evidence is not appropriate in this case. Computing the MPE involves assigning a value to every property in the failure dependency graph to maximize the probability of the observed evidence. In Internet fault diagnosis, however, a user requesting fault diagnosis does not know nor care about all components and properties in the network; typically they only want to know which of a small number of possible causes of failure is most likely.

7.2 Data structures

A key feature of the procedure that agents follow for diagnosis is that it is dynamic and not domain-specific so that agents can take into account new information and knowledge received from new agents. Therefore a CAPRI agent generates the data structures for fault diagnosis dynamically each time it receives a diagnostic request.

CAPRI agents represent the information they have about network component individuals and component dependencies using several types of data structures. For every diagnostic request, an agent constructs a *component graph* representing observations and beliefs about components relevant for the current diagnosis, and a *failure dependency graph* derived from the component graph and dependency knowledge that represents the probabilistic evidence and dependencies for properties of components in the component graph. The component graph represents the current state of diagnosis, including the results of tests that have been performed, the observations received from other agents about the current failure, and the relationships between components and diagnostic tests. An agent uses its component graph to determine whether it can respond to a request using cached information. Whenever the component graph changes or new dependency knowledge becomes available in the process of diagnosing a failure, an agent rebuilds its failure dependency graph to take into account new evidence or dependencies. The failure dependency graph contains the probabilistic dependencies and evidence that an agent uses to infer beliefs about the value of unobserved properties and to compute the value of performing diagnostic actions. I describe these data structures in more detail below.

Each diagnostic agent also maintains a *component information base* of cached information obtained from previous diagnoses and a *dependency knowledge base* of dependency knowledge. The component information base stores observations, beliefs, and likelihoods received from other agents so that this information may be used for diagnosing future failures. For example, an agent can store information about failed web servers received from other agents so that it can quickly diagnose future connection failures to those servers. The dependency knowledge base stores probabilistic dependency knowledge about components. An agent can add to its dependency knowledge base by learning probabilistic dependencies from observations or by incorporating knowledge from other agents. For example, an agent might learn the conditional probability a ping test succeeds given the status of the destination host and store this knowledge in its dependency knowledge base.

7.2.1 Component information base

A component information base stores cached information about component individuals from the agent's component graph. The component information base contains a list of observations, beliefs, and likelihoods about components, indexed by their identifying properties. Each entry in the component information base also includes an expiration time. An agent's component information base might contain multiple observations of the same component, each made at a different time, from different agents, and containing information about a different set of properties. Some of these observations, beliefs, and likelihoods may conflict with one another. Agent can resolve such conflicts using the metadata associated with pieces of information, preferring more recently made observations to older observations, for example.

Caching information in a component information base helps support aggregation of requests to reduce the communication costs of diagnosis. Aggregation of requests refers to the ability to diagnose multiple similar failures using the same information. In CAPRI, aggregation of requests can occur whenever a single diagnosis provider agent offers to diagnose failures for many other agents. If the failures that the requesters want diagnosed tend to be similar, then the diagnosis provider agent will likely be able to use the same data to diagnose multiple failures at relatively low cost. For example, if each Internet Autonomous System (AS) has an associated diagnostic agent that can diagnose failures within that AS, and each agent only accepts requests from agents for neighboring ASes, then diagnosing a failure that may be caused by a failure of any one of the ASes along an IP path by requesting diagnosis from the agent for the next AS hop will result in efficient aggregation. By returning cached information from its component information base, a diagnostic agent can answer multiple similar requests without conducting additional tests or requests.

7.2.2 Dependency knowledge base

A knowledge base data structure stores the probabilistic dependency knowledge that an agent receives from other agents. A knowledge base entry specifies a conditional probability table for a component class, property, and set of parent variables. Each piece of dependency knowledge provides a way to infer the value of a property based on its parent variables. See Section 4.2.4 for more details about dependency knowledge.

Each piece of dependency knowledge also has an associated expiration time. Expiring knowledge forces agents to periodically reload dependency knowledge from the knowledge provider agent, allowing agents to discover updated dependency knowledge when available.

7.3 Building a component graph

The first step of fault diagnosis is parsing the information provided in a diagnostic request to construct a component graph. This involves identifying the components and properties described in the body of the request and correctly setting observations, beliefs, and relation-

ship properties for each component. An agent also adds any new dependency knowledge to its knowledge base. In addition, an agent retrieves component class and property definitions for any new components and properties.

Agents parse diagnostic messages to update two data structures: a component information base of known component individuals and diagnostic test results, and the associated beliefs of the component properties; and a dependency knowledge base containing probabilistic dependency information for each component and diagnostic test class. Using these two data structures, an agent can perform probabilistic inference to infer the status of components in the component model.

The procedure for parsing messages is as follows:

1. An agent receives a message.

2. The agent decodes the message according to the diagnostic message format. That is, it parses the XML message to determine whether it is a diagnostic request, a notification, or a diagnostic response. The agent then extracts the headers and body according to the message definition.

3. The agent determines whether it has the capability to answer the message, or if the message is malformed, expired, or if for some other reason the agent cannot return a response. If the agent cannot respond to the request, it returns an error message.

4. The agent parses the body of the message, extracting observations, beliefs and likelihoods, and knowledge, as well as the input component if it exists.

5. For each observation:

   (a) First, determine whether it describes a component or a test result based on the element name inside the observation. Suppose it is a component.

   (b) Next check each `class` element inside the `component` element. If a class definition is not in the agent's class definition table, the agent retrieves the component class definition using the URI for the class.

   (c) The agent parses each property of the component. If the property definition for any of the properties is not in the agent's property definition table, it retrieves the appropriate definition using the URI of the property.

   (d) The agent checks its component graph to determine whether this component is already in its component graph based on the component's identifying properties or based on the component ID.

   (e) The agent adds or updates the information in its component graph as necessary based on the metadata associated with the observation. Note that it may need to make two passes: one pass to create all the components described in the message, and another pass to make the proper assignments to the component relationship properties. If an agent has multiple observations of the same component, it may keep multiple conflicting observations or it may discard one or more observations based on its local conflict resolution policy.

6. For each belief or likelihood:

   (a) Extract the metadata for the belief or likelihood.

   (b) Identify the component individual specified in the subject using either its identifying properties or its component ID. If the agent's component model does not contain the specified individual, then create the individual and set its properties accordingly.

   (c) Identify the property of the component specified in the belief or likelihood.

   (d) Identify the evidence on which the belief or likelihood is based.

   (e) Beliefs and likelihoods are expressed as probability tables. Parse the distribution or likelihood table to produce a probability table. Associate this table with the component property in the component graph. In case of multiple conflicting observations, beliefs, or likelihoods for the same property, the agent decides which to use or discard based on the metadata. An agent may choose to keep multiple conflicting beliefs or it may discard one or more beliefs or likelihoods based on its local conflict resolution policy.

7. For each piece of dependency knowledge:

   (a) Extract the metadata from the knowledge.

   (b) Identify the component class specified in the subject. If the agent's knowledge base does not contain the specified class, then create a new entry for the class.

   (c) Identify the property of the component specified in the knowledge.

   (d) Identify the parent paths for the knowledge.

   (e) Parse the CPT. First, identify the parent properties. Use the `parentVal`, `value`, and `p` elements to construct a CPT. Store this CPT in the agent's knowledge base indexed by the component class (from step (b)), property (from step (c)), and parent paths. In case of multiple conflicting CPTs for the same component class, property, and parents, the agent may keep multiple CPTs or it may decide to discard one or more CPTs based on its local conflict resolution policy. For example, it may choose to keep only the most recent piece of knowledge.

This procedure allows agents to combine information from multiple sources into a single component graph. For example, consider a regional diagnostic agent that receives two observations from different sources. First it receives an observation in a diagnostic request from a user agent about an *HTTP Connection* and its associated *Local Network* and *DNS Lookup* components. Next the regional agent requests additional observations from a DNS lookup test specialist agent. The DNS lookup test agent provides the regional agent with an observation describing the results of a *DNS Lookup Test* for a *DNS Lookup* component. Figure 7-2 illustrates the information in these two observations. Using the procedure for parsing diagnostic information, an agent identifies components that are the same and combines these observations into a single component graph, illustrated in Figure 7-3. This example illustrates a relatively straightforward case of unifying information from multiple

sources in which the observations share only one component in common and there are no conflicts, but in general there may be more overlapping components and conflicting observations.

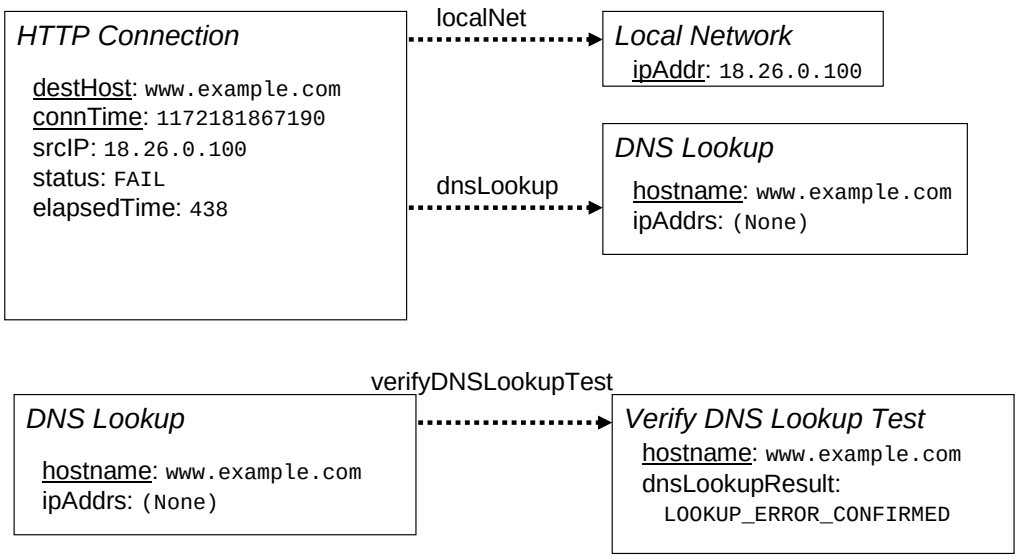

Figure 7-2: An agent obtains diagnostic observations from two different sources. Note that both observations refer to the same *DNS Lookup* component.

Note that an agent can disambiguate multiple components of the same class using relationship properties and property paths. For example, consider a *DNS Lookup* component *D* with two relationship properties, `primaryDNSServer` and `secondaryDNSServer`, each referring to a different *DNS Server* component. Suppose that agent *A* has an observation of *DNS Lookup* component *D* in its component graph but does not know the identity of the two *DNS Servers* referred to by the `primaryDNSServer` and `secondaryDNSServer` relationship properties. Agent *A* wishes to request an observation of the status of the `primaryDNSServer` component from another agent *B*. Using property path notation, agent *A* requests an observation of `primaryDNSServer.status`, providing the *DNS Lookup* component *D* as the input component. Agent *B* returns a response indicating that the *DNS Server* component to which the `primaryDNSServer` relationship property of component *D* refers has status `OK`. Thus when agent *A* receives the observation from agent *B*, it can unambiguously determine which *DNS Server* component the observation refers to.

7.4 Preprocessing

After constructing a component graph, an agent may perform certain preprocessing operations on the information in the component graph before performing additional actions. For example, for privacy or policy reasons, an agent may encode or remove information before

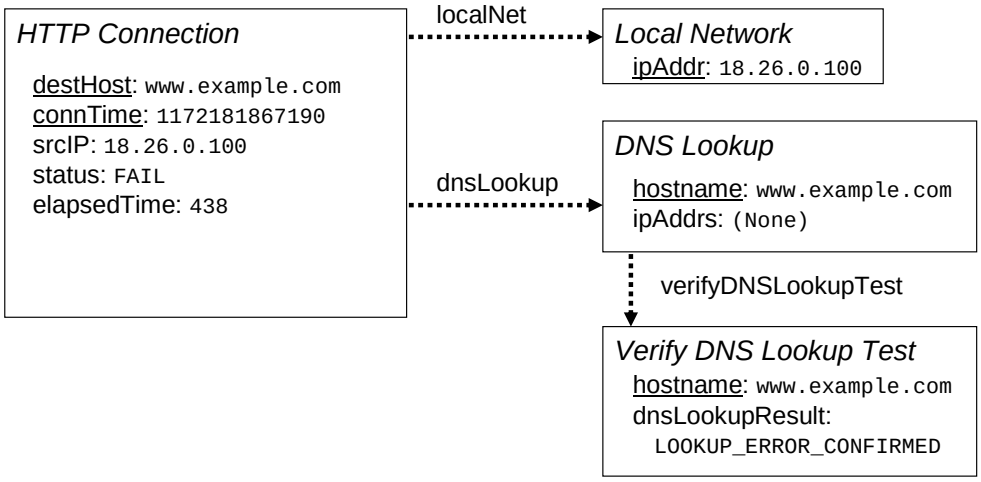

Figure 7-3: An agent unifies the pieces of diagnostic information from Figure 7-2 into a single component graph.

taking additional actions. In addition, in some cases an agent may modify information in the component graph to correct errors due to unsynchronized clocks, for example.

7.5 Incorporating cached information

To reduce the costs of diagnosis, agents can incorporate previously cached information from the component information base into the component graph. For each component in its component graph, a diagnostic agent determines whether it has previously cached information about that component and sets the appropriate properties and beliefs in its component graph. Cached information may include observations such as the results of recent tests, beliefs such as recently inferred status of components, and relationships such as the previously computed dependencies of an AS path. For example, an agent may set the status of an AS path based on cached information from a previous diagnosis. An agent may cache component relationships in its component information base as well. For example, suppose an agent has an entry in its component information base stating that the *IP Routing* component with `srcIP` 18.26.0.100 and `destIP` 140.211.166.81 has a relationship property `asPath` referring to the *AS Path* component with `srcAS` 3 and `destAS` 3701. An agent may then use this component information base entry to set the `asPath` relationship property of the *IP Routing* component in its component graph and create the corresponding *AS Path* component.

Caching information in a component information base can greatly reduce the cost of diagnosing multiple failures. For example, an agent that determines that an HTTP connection failure is due to a failed web server can use this information to diagnose future failures to the same destination without performing any additional tests or requests.

One challenge is determining whether or not to use cached information. Using cached

information can reduce the cost of diagnosis, but it may also reduce accuracy if the information is out of date. Deciding whether to use cached information requires considering several factors. Different types of information may have different lifetimes; some observations such as the AS number of an IP address may be valid for years, while others such as the average round-trip time of a TCP connection may change rapidly. Also, the consequences of using out-of-date information may differ greatly. Incorrectly identifying an AS hop in the AS path of an IP route may not matter much if the status of the incorrect AS hop is the same as the status of the actual AS hop. On the other hand, if an agent incorrectly believes a web server to have failed when it has not, it may mistakenly diagnose an HTTP connection failure. To address this challenge, the originator of a piece of information sets the expiration time of every piece of information it produces. If a piece of information has expired, then an agent does not use the information and removes it from its component information base.

7.5.1 Incorporating past evidence using DBNs

Agents may also use cached data in a component information base to infer the current status of a component using a temporal failure model described by probabilistic dependency knowledge. This means that if a diagnostic agent has knowledge of how long a failure will last, it can communicate this information to other agents that can then use this knowledge to infer the status of a component from past evidence. Agents may model temporal dependencies among components using dynamic Bayesian networks (DBNs) [78] to incorporate cached data. A DBN enables an agent to infer the status of a component from past evidence given a discrete time hidden Markov model (HMM) of the network.

A Markov model encodes the probability that a variable has a particular value given past values of the variable. Many network components can be modeled as Markov processes. For example, a Gilbert model is a common model of network link failures in which the status of a link depends probabilistically on its status in the previous time step [96].

One can transform a HMM into a dynamic Bayesian network (DBN) by representing the status of a component in each time step with a different variable. For example, in a discrete first-order Markov model, the status X_i of a component X at time i depends probabilistically on its status in the previous time interval $i-1$. An agent represents the transition probabilities $P(X_j|X_{j-1})$ as a conditional probability table. To incorporate k time steps of past evidence into this model, an agent "unrolls" this DBN by adding additional nodes and edges. It can then compute the marginal probability $P(X_i|o)$ of component X_i given observed data o using standard inference techniques:

$$P(X_i|o) = \sum_{X_{i-1},\ldots,X_{i-k}} P(X_{i-k}|o) \prod_{k<j\leq i} P(X_j|X_{j-1}) \tag{7.2}$$

For greater accuracy an agent can model network components using higher order hidden Markov models using the same approach.

One challenge of reasoning in DBNs is that if there are many temporal dependencies in the model, the cost of inference can increase exponentially as the number of time steps increases. Fortunately, some properties do not have temporal dependencies. For example, the

status of abstract components such as *TCP Connections* and *DNS Lookups* do not have any direct temporal dependencies since they depend only on the present status of the underlying hardware such as routers and DNS servers. This can greatly reduce the cost of inference and enable agents to make use of past evidence to reduce the costs of data collection and communication.

A DBN enables an agent to infer the most likely status of a component given past evidence of the status of the component. Note that this also enables an agent to compute the relevance of a past observation with respect to a set of other variables given the available evidence [61]. An agent might compute the relevance of information to decide whether to use cached information or when to expire information in its cache.

7.6 Constructing a failure dependency graph

After incorporating cached information into its component graph, an agent constructs a failure dependency graph from the component graph and dependency knowledge. A failure dependency graph is a Bayesian network that describes components and diagnostic tests involved in the failure, any applicable observations and beliefs about these components, and the probabilistic dependencies among properties of these components and tests. More formally, a failure dependency graph consists of variables, edges, probabilistic dependency knowledge, and evidence, where each variable represents a property of a component individual and each edge (u, v) indicates that the value of the parent variable u influences the value of the child variable v. A failure dependency graph may incorporate both deterministic evidence about a variable from an observation as well as probabilistic evidence about a variable from a belief or likelihood. Each variable also has probabilistic dependency knowledge specifying the conditional probability of the variable given the value of its parent variables. A failure dependency graph can be thought of as a type of probabilistic relational model [33].

An agent uses a failure dependency graph to make probabilistic inferences to diagnose a failure. For example, a diagnostic agent with the component graph in Figure 7-3 and dependency knowledge from Table 7.1 can construct the failure dependency graph in Figure 7-4. This failure dependency graph illustrates that the status of the *HTTP Connection* to www.example.com at time 1172181867190 depends on the status of a *Local Network*, *HTTP Server*, *IP Routing*, and *DNS Lookup*. Additionally, the result of a *Verify DNS Lookup Test* can provide evidence about the status of the *DNS Lookup*. This agent has evidence that the *HTTP Connection* has failed and that the result of the *Verify DNS Lookup Test* is `LOOKUP_ERROR_CONFIRMED`.

To construct a failure dependency graph, an agent consults its dependency knowledge base to identify relevant pieces of dependency knowledge for the components in its component graph. At the start of diagnosis, an agent begins with a failure dependency graph consisting only of the failed component and the list of candidate explanations. For each component property P whose value is unknown, an agent may be able to infer its status either from dependency knowledge describing how the property P depends on other component properties (causal inference), from dependency knowledge describing how the properties of other components and tests depend on the status of P (diagnostic inference),

HTTP Connection status:

localNet.-status	httpServer.-status	dnsLookup.-status	ipRouting.-status	$P(\text{status} = \text{OK} \mid \text{parents})$
OK	OK	OK	OK	1
OK	FAIL	OK	OK	0
FAIL	OK	OK	OK	0
FAIL	FAIL	OK	OK	0
OK	OK	FAIL	OK	0
OK	FAIL	FAIL	OK	0
FAIL	OK	FAIL	OK	0
FAIL	FAIL	FAIL	OK	0
OK	OK	OK	FAIL	0
OK	FAIL	OK	FAIL	0
FAIL	OK	OK	FAIL	0
FAIL	FAIL	OK	FAIL	0
OK	OK	FAIL	FAIL	0
OK	FAIL	FAIL	FAIL	0
FAIL	OK	FAIL	FAIL	0
FAIL	FAIL	FAIL	FAIL	0

Local Network status:

$P(\text{status} = \text{OK})$
0.95

HTTP Server status:

$P(\text{status} = \text{OK})$
0.99

DNS Lookup status:

$P(\text{status} = \text{OK})$
0.98

IP Routing status:

$P(\text{status} = \text{OK})$
0.99

Verify DNS Lookup Test dnsTestResult:

dnsLookup.status	$P(\text{dnsTestResult} = \text{LOOKUP\_ERROR\_CONFIRMED} \mid \text{parents})$
OK	0.01
FAIL	0.80

Table 7.1: An agent's dependency knowledge base contains both probabilistic and deterministic dependency knowledge about component classes and properties.

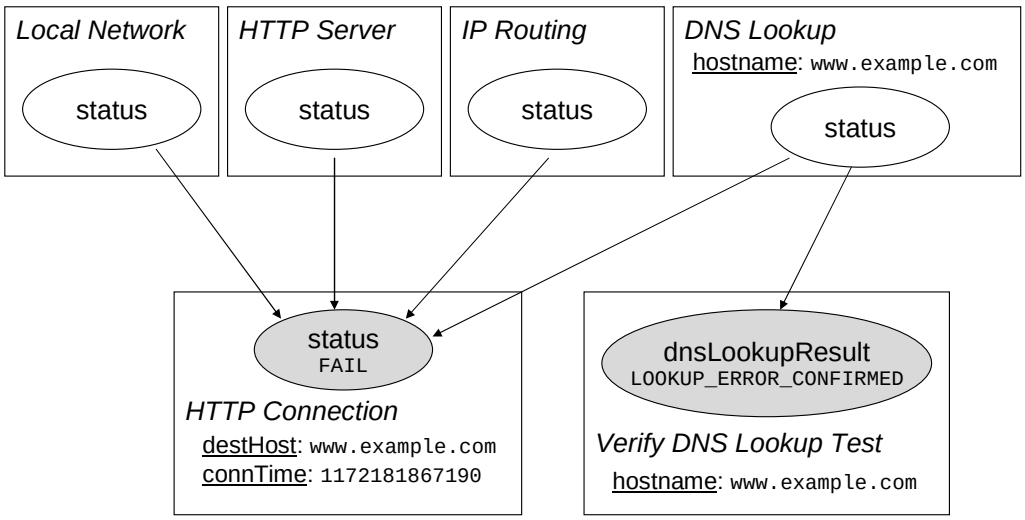

Figure 7-4: An agent constructs a failure dependency graph using the component graph from Figure 7-3 and dependency knowledge from Table 7.1

or from knowledge of prior probabilities. For each piece of dependency knowledge that the agent wishes to apply, it adds additional components or diagnostic tests to its failure dependency graph as necessary. For example, in order to infer the status of an HTTP server using diagnostic inference an agent may choose to add a *Ping Test*.`pingTestResult` variable to its failure dependency graph and add an edge from *HTTP Server*.`status` to the new variable. For each additional component or diagnostic test added to the failure dependency graph, an agent checks its component information base to determine whether it already has observations or beliefs about any of the properties of these components or tests. Note that at this point the agent does not yet actually perform any of the diagnostic tests; it is only constructing a failure dependency graph to determine what additional diagnostic actions are possible. An agent chooses what dependency knowledge to use based on its expected cost, expected accuracy, or local policy. An agent recursively adds additional components to its failure dependency graph. For each unknown component property, if an agent has dependency knowledge that it can use to infer the value of that property, it may add additional components to its failure dependency graph. For example, to infer the value of an *HTTP Connection*.`status` variable, an agent may add *IP Routing*.`status` variables to its failure dependency graph. To infer the value of an *IP Routing*.`status` variable, it may then add additional *AS Path*.`status` variables, and so on.

7.7 Inferring a diagnosis

The procedure for diagnosis in CAPRI allows each agent to set a confidence threshold for diagnosis and provide a set of candidate explanations *H* to test. This gives users and agents

some control over the accuracy and cost of diagnosis. The higher the threshold, the more accurate the diagnosis and the greater the potential cost. In addition, a confidence metric lets agents indicate the probability that their diagnosis is correct. If the Bayesian network used for inference is correct, the confidence threshold gives a lower bound on the accuracy. Note that when dependencies are not deterministic, it may not be possible to achieve 100% confidence in a diagnosis.

The set of hypotheses that an agent must distinguish between also affects the cost of diagnosis. Generally if there are only a small number of candidate explanations, an agent can determine which is most likely at fairly low cost. On the other hand, if the set of candidate explanations is large and involves many variables, then diagnosis may be more costly. For example, a requester may simply wish to confirm a suspicion that a component `status` property X has the value `FAIL`. That is, the set of candidate explanations is

$$H = \{\{X = \texttt{OK}\}, \{X = \texttt{FAIL}\}\} \tag{7.3}$$

Another possibility is that the requester wants to know the value of all m parent component `status` variables $\mathbf{Y} = \langle Y_1, \ldots, Y_m \rangle$ for a component `status` variable X:

$$H = \{\{\mathbf{Y} = \mathbf{v}\} | \mathbf{v} \in \{\texttt{OK}, \texttt{FAIL}\}^m\} \tag{7.4}$$

If the failed component depends on all on its parent components functioning, a requester may simply wish to know which of the dependencies is most likely to have failed:

$$H = \{\{Y_1 = \texttt{FAIL}\}, \ldots, \{Y_m = \texttt{FAIL}\}\} \tag{7.5}$$

Note that if the set of variables to consider in each candidate explanation differs, then multiple candidate explanations may be true.

Giving the user control over the confidence threshold and the set of candidate explanations to consider allows them to choose between a quick and cheap diagnosis and a slow and accurate one. Diagnostic agents under load may also choose to lower their confidence threshold or the number of candidate explanations they consider to reduce the cost of diagnosis. In addition, by adjusting the set of candidate explanations a user can initially request diagnosis using a small set of candidate explanations to get a quick, low-cost response, and then request more detailed diagnoses later if necessary.

Alternatively, one can also consider diagnosis using a procedure to maximize accuracy without exceeding the budget. Krause and Guestrin describe algorithms to address these issues in more detail [51].

7.8 Performing diagnostic actions

If an agent cannot infer the root cause of a failure with sufficient confidence, it repeatedly performs additional diagnostic actions until it has sufficient confidence in its diagnosis. CAPRI differs from previous architectures for fault diagnosis that only support fixed communication patterns among diagnostic agents. Instead, CAPRI agents dynamically compute the set of available next actions from service descriptions and select an action based

on the value of each action. Diagnostic actions may include both local tests and requests for additional information from other agents. After each diagnostic action, an agent updates its failure dependency graph to take into account the new observations and evidence it collects from other agents. Note that the set of available actions may change as agents learn new dependency knowledge, researchers develop new diagnostic tests, and new agents advertise their services. Therefore agents need a way to dynamically determine what actions are possible and the value of those actions while being able to set a policy to prefer certain actions based on its preferences, requirements, and capabilities.

An agent may have the ability to perform both local and remote actions. One type of local action is to fill in the property values of components using local information. For example, a local agent might be able to determine the IP address of the current user or the time of the last HTTP connection using local information from applications or the operating system. Another local action is to perform a diagnostic test to determine the properties of a diagnostic test component. For example, an agent might conduct a *Ping Test* in order to obtain evidence of a ping test result to infer the status of an *IP Host*. An agent may also request observations, beliefs or likelihoods, or dependency knowledge from other agents.

Not all actions are relevant for diagnosis, however. The set of useful next actions an agent may take depends on the services available to it as well as the information in an agent's component graph. To identify the set of useful next actions that it can take, for each service in its service table, an agent determines whether it has any components in its component graph that can be used as an input for that service. In addition, it determines whether that service would produce useful outputs given the input component. If all these criteria are met, the agent computes the value of the service and adds the service and input to its list of available possible next actions.

Each action may have a different value for diagnosis. The challenge is for the agent to select an action that significantly improves the accuracy of its diagnosis while minimizing the cost expended. An agent takes into account several factors to calculate service value. Firstly an agent takes into account the myopic value of information. That is, what is the expected value of having the information that the service produces for diagnosing the current failure. One way to quantify this value is to compute the expected change in diagnostic confidence.

To compute the myopic value of information for an action given a service and an input component, an agent first determines what evidence that service will produce based on the service description in the agent's service table. The evidence that the service produces includes all its output observations and all the evidence on which its output beliefs are based. Next, for each of these pieces of evidence, an agent computes the expected change in diagnostic confidence of having that evidence. Simply computing the myopic value of information may not be adequate, however, since the value of conducting a series of tests may be different from the sum of the value of the individual tests. If two tests have similar value but are based on the same evidence, it may only be useful to perform one of the two tests. Other researchers have considered the problem of selecting actions non-myopically [51, 43, 58], but algorithms for computing the non-myopic value of information involve much more computational complexity or require domain-specific assumptions for efficient approximation.

Another factor to take into account is the expected future distribution of failures. For

the purposes of aggregation, it is important to select actions that produce information that can be reused to cheaply diagnose future failures. This requires an agent to predict the expected distribution of future failures.

Another consideration is whether performing an action enables an agent to make use of cached information in its component information base. For example, an agent may have cached information about the status of AS paths. By performing an action that reveals the AS path of an IP route, an agent may be able to make use of cached information in its component information base.

Once an agent has a list of possible next actions and has computed the value of each of these actions, it sorts these actions by utility, where utility is a function of value and cost. An agent may choose a utility function to use depending on the agent's tradeoff between accuracy and cost. This is similar to the way in which the service selection agent in the Personal Router chooses which Internet access service provider to use [56]. Finally the agent performs the action with greatest utility and repeats the steps above until its confidence exceeds a threshold, the request expires, the budget is expended, or no more actions are available. Each time an agent performs an action, it removes that action from its set of available actions. To prevent endlessly retrying the same service, an agent only recomputes the set of available next actions if it receives new information.

Note that this procedure can automatically work around diagnostic agent failures; if a specialist agent cannot perform a diagnostic test or is not reachable, and another specialist agent with similar capabilities is available, the regional agent will automatically try requesting information from the next specialist agent. On the other hand, if multiple specialists all provide the same information, then after getting the information from one specialist, the regional agent knows that requesting information from the other specialists provides no additional diagnostic value and so will not request information from the other specialists. This action selection procedure also provides agents with a mechanism for providing redundant backup services. If one or more agents advertise services with identical inputs and outputs but different costs, then a requester will always prefer the lower cost service if available; if that service fails or becomes unavailable, however, then it will use the backup instead. Agents can also use a similar technique to preferentially handle requests from certain other agents. For example, a regional agent that wishes to handle diagnostic requests for all agents in a particular AS can advertise its service at a lower cost than other generic regional agents that handle requests from all agents. Thus a user agent choosing a regional agent to contact will choose the regional agent for its AS if available, otherwise it will choose one of the generic regional agents.

This procedure for action selection gives agents great flexibility in how they perform diagnosis. The operator of a diagnostic agent can adjust several parameters to change how diagnosis is performed. For example, to avoid costly diagnostic actions, one can use a utility function that penalizes actions with high cost. Similarly, one can tune the parameters of the service value computation to favor short-term accuracy or to minimize long-term cost. This thesis does not discuss the range of possible adjustments one can make, but it is important to note that this architecture supports a wide range of action selection strategies. Other researchers describe some possible approaches to select diagnostic actions [76, 58, 51, 3, 61, 62]. The probabilistic approach to computing action value and selecting actions presented in this thesis resembles approaches described in previous work,

but whereas previous work mainly focuses on action selection for domain-specific diagnosis, this thesis addresses the challenges of selecting actions in a general way using service descriptions.

This procedure also supports distributed diagnosis using belief propagation, in which multiple agents in different parts of the network with knowledge of different components perform probabilistic inference and then share their beliefs and likelihoods to compute a diagnosis in a distributed manner. To perform belief propagation in CAPRI, an agent requests beliefs and likelihoods from other agents and performs probabilistic inference. The advantage of belief propagation is that each agent only needs to know a subset of the complete dependency graph and can compute beliefs using only local information. Procedures for belief propagation in Bayesian networks for distributed inference are well known. For more details please refer to [71].

7.9 Returning a response

After completing all the necessary data collection and diagnostic inference, an agent assembles the requested information into a response message. The provider agent adds to the response message the output observations, beliefs, likelihoods, knowledge, and diagnosis that the requested service offers.

7.10 Postprocessing

After performing a diagnosis, an agent propagates evidence and beliefs to other agents based on notification subscriptions. Such propagation of evidence and beliefs can potentially reduce the future costs of diagnosis. For example, an agent that observes a failure in a critical network component may communicate this observation to other agents that can then use this information to diagnose future failures. Agents can use service advertisements to construct an aggregation-friendly topology to facilitate such aggregation (see Chapter 5).

Chapter 8

Prototype Diagnostic Network

The previous chapters described the CAPRI architecture for fault diagnosis and the benefits it can provide in terms of extensibility, cost management, and ability to deal with incomplete information. This chapter demonstrates these benefits of CAPRI for real diagnostic agents. To evaluate the capabilities of CAPRI, I develop a prototype network of agents for diagnosing HTTP connection failures for end users. This prototype implementation includes several types of agents, including user agents, regional agents, a knowledge agent, and several types of specialist agents that perform a wide range of active and passive tests, including DNS lookup tests, connectivity probes, Rockettrace measurements, and web server history tests. I show how the CAPRI component ontology and communication protocol enables heterogeneous agents to discover the capabilities of other agents and communicate diagnostic information about new components and diagnostic tests. I show how agents in this network can reduce the probing and communication costs of diagnosis by aggregating related requests and propagating evidence and beliefs. I also demonstrate the ability to add new diagnostic agents to the system by implementing a CoDNS lookup test agent. Finally, I show how agents can perform probabilistic diagnosis using whatever evidence is available to deal with incomplete information.

This chapter describes the types of failures agents diagnose in this prototype implementation, the types of agents I implement, the classes of diagnostic components and tests agents understand, the types of information agents communicate, and the procedure that agents use for diagnosis. Finally I describe the advantages of fault diagnosis using this prototype implementation over previous research in distributed fault diagnosis. The next chapter presents the results of experiments to quantify the benefits of diagnosis using the CAPRI architecture.

8.1 Diagnosis of HTTP connection failure

For my prototype implementation of fault diagnosis, I choose to diagnose HTTP connection failures for several reasons. Firstly, an HTTP connection failure can have multiple causes, and well-known diagnostic tests exist for testing these causes. Secondly, web browsing is a commonly used network application for which it is relatively easy to collect data from users. In this experiment, diagnostic agents determine, for each HTTP connection failure

observed by a user's web browser, whether the failure was due to a DNS lookup failure, a web server failure, a network connectivity problem from the user to their ISP, or whether an IP routing failure[1] occurred between the user's ISP and the destination web server. Diagnosis in this experiment occurs in real time. When the user agent detects a failure, it requests diagnosis and displays the result to the user when diagnosis is complete.

Note that it is not always possible to determine the true cause of failure. The goal of this prototype is not to diagnose failures with maximum accuracy, but rather to demonstrate the benefits of the CAPRI architecture in terms of extensibility, the ability to perform distributed diagnosis, and the ability to control the cost of diagnosis.

8.2 Types of agents

Firstly, I show that CAPRI allows heterogeneous agents with different capabilities in different parts of the network to cooperate to diagnose failures. Diagnosis involves conducting diagnostic tests, aggregating data, learning dependency knowledge, and probabilistic inference. In the prototype implementation I develop, no single agent performs all of these functions; instead, for improved scalability and cost management I deploy several types of diagnostic agents throughout the Internet. User diagnostic agents implemented as Mozilla Firefox extensions run inside users' web browsers and collect observations about failures.[2] Regional diagnostic agents respond to diagnostic requests from user agents and act as aggregation points to reduce the load of diagnosis on specialist agents. Regional agents can request additional diagnostic information from four types of specialist agents: stats agents, web server history agents, AS path test agents, and DNS lookup test agents. Stats agents collect user connection history information to destination hostnames and provide observations of various statistics about connections to the hostname. Web server history agents collect user connection history information to web server IPs and infer web server status. AS path test agents test the status of IP routing along the AS hops from the user to the web server. DNS lookup test agents test the correctness of the user's DNS lookup. Knowledge agents collect aggregate data from users and server history agents to produce new dependency knowledge and provide this knowledge to other agents. All agents advertise and retrieve service descriptions from a centralized service directory. The specialist agents and regional agents operate on Planetlab at sites around the world. The learning agent and the service directory run on a machine at MIT. Appendix B contains the ontologies, service descriptions, and knowledge for these agents.

This prototype implementation supports the addition of new agents as well. New user agents may join at any time. When new user agents join the network for fault diagnosis, they automatically look up the identities of the regional agent and the server history agents with which they need to communicate using the service directory. The directory server assigns regional agents to users based on their AS to distribute load and provides regional agents with a way to locate the specialist agents they wish to contact. For ease of

[1] Technically, the failure of an IP path may be caused either by an incorrect route (an IP routing failure) or a failure to forward a packet along the route (an IP forwarding failure). It is difficult for users to distinguish among these two cases, and so in this thesis I refer to both types of failures as IP routing failures.

[2] http://servstats.mozdev.org/

implementation and evaluation, this prototype implementation uses a centralized directory server. This directory server provides agents with a list of available diagnostic services that other agents offer, including knowledge, observation, and belief services as well as subscription notifications. An agent stores this list in its service table for future lookups. If additional robustness is desired, this centralized directory service could be replaced with a more distributed peer-to-peer system, but implementing such a directory service is outside the scope of this thesis.

This prototype implementation includes over 10,000 user agents from around the world, 4 web server history agents, 4 stats agents, 3 DNS lookup test agents, 14 AS path test agents, and 15 regional agents.

8.3 Components and diagnostic tests

In this implementation, agents exchange diagnostic information according to the component ontology described in Chapter 4 and using the protocol described in Chapter 6. Agents communicate information about several types of components and diagnostic tests. In this section I briefly describe each of the types of components and tests that agents understand. Section 8.5 describes how agents observe these components and properties and use this information for diagnosis. The complete component ontology can be found in Appendix B.

8.3.1 Components

Agents in this prototype implementation diagnose HTTP connection failures. An *HTTP Connection* class has descriptive properties `srcIP`, `destHost`, `destHash` (MD5 hash of the destination hostname), `connTime`, and `status`; and relationship properties `dnsLookup`, `httpServer`, `localNet`, and `ipRouting`. An *HTTP Connection* is identified by `connTime` and `destHash`. An *HTTP Connection* failure may be due to a failure in the *DNS Lookup*, a failure at the destination *HTTP Server*, a *Local Network* connectivity problem between the user and their ISP, or an *IP Routing* failure elsewhere in the network. When an agent starts up for the first time, it retrieves dependency knowledge about *HTTP Connections* from the knowledge agent.

An *HTTP Server* class has an identifying property `ipAddr` indicating its IP address. An *HTTP Server* also has a status property and an aggregate property `consecFailuresToServer` representing the number of consecutive users who experienced failures connecting to this server. Server history agents can probabilistically infer the status of a server from `consecFailuresToServer`.

A *Local Network* class has an identifying property `ipAddr`, a `status` property, and an aggregate property `consecFailuresFromUser` representing the number of consecutive unique destinations to which this host experienced HTTP connection failures. User agents and regional agents can use `consecFailuresFromUser` to infer the status of a *Local Network* component.

A *DNS Lookup* class has a `status` property, a destination `hostname` property, and an IP address list property `ipAddrs`. The IP address list property contains a list of the IP

addresses for the given hostname returned by the DNS lookup, or is empty if the DNS lookup failed.

An *IP Routing* class has a `status` property, a `srcIP` property, and a `destIP` property. *IP Routing* components are identified by their source and destination IP addresses. The status of an *IP Routing* component depends on the status of the underlying *AS Path*.

AS Path components have a `srcAS` property, a `destAS` property, a `status` property, and two component relationship properties: `nextASHop`, and a `nextASPath` property that refers recursively to another *AS Path*. *AS Hop* components also have the properties `srcAS`, `destAS`, and `status`.

8.3.2 Diagnostic tests

To illustrate the range of possible diagnostic tests CAPRI supports, I implemented several different types of diagnostic tests that vary greatly in terms of required inputs, outputs, and cost. These diagnostic tests include both passive measurements computed from historical observations, as well as active measurements that probe the current state of the network. In addition, some diagnostic tests such as a *Verify DNS Lookup Test* produce only an observation of the value of the property of a diagnostic test. Others also produce information about the dependencies of a component, such as an *AS Path Test* that provides information about both the AS path dependencies of an IP path as well as information about the status of the links along the path.

One diagnostic test is to observe the error code returned by the web browser application. A *Firefox Error Test* has a descriptive property `ffoxErrorCode` and a relationship property `httpConn` indicating the corresponding *HTTP Connection*. See the Appendix B for the full list of error codes.

Another useful diagnostic test is to determine whether the user can connect to a known good web server. An *Outbound Conn Test* diagnostic test class has descriptive properties `srcIP`, `probeURI`, and `probeResult`. The test result is either `OK` or `FAIL` depending on whether the user agent was able to connect to the destination host. Users and regional agents can use the result of an *Outbound Conn Test* to infer the status of a *Local Network*.

DNS specialist agents can perform a *Verify DNS Lookup Test*. This test has descriptive properties `hostname` and `dnsLookupResult`; and a component relationship property `dnsLookup` referring to the DNS Lookup component tested. The test result may be one of six possible values:

LOOKUP_ERROR_CONFIRMED Neither the user nor the DNS agent was able to get an IP address for the given hostname.

LOOKUP_ERROR_UNCONFIRMED The user was unable to get an IP address, but the DNS agent was able to obtain an IP address for the given hostname.

CORRECT The IP addresses in the DNS Lookup component match the IP addresses obtained by the DNS agent.

INCORRECT The IP addresses in the DNS Lookup component differ from the IP addresses obtained by the DNS agent.

LOOKUP_ERROR The user obtained an IP address for the hostname, but the DNS agent failed to get an IP address.

ALIAS The DNS agent's DNS lookup indicated that the hostname is an alias for another hostname. In such situations, frequently different users will obtain different IP addresses for the same hostname. For example, content delivery networks such as Akamai use DNS aliases to return different IP addresses to different users.

A DNS specialist agent infers the status of a *DNS Lookup* using the value of `dnsLookupResult`.

IP routing specialist agents can perform *AS Hop Tests* and *AS Path Tests* to determine the status of *IP Routing* between two Internet autonomous systems. An *AS Hop Test* has descriptive properties `srcAS`, `destAS`, and `asHopTestResult`; and a component relationship property `asHop` indicating the *AS Hop* being tested. An `asHopTestResult` is either `OK` or `FAIL`, depending on whether the agent was able to successfully route IP packets from the source AS to the destination AS. The source and destination ASes of an *AS Hop Test* must be neighboring ASes. An *AS Path Test* has descriptive properties `srcAS`, `destAS`, and `asPathTestResult`. An `asPathTestResult` is either `OK` if the traceroute was able to reach the AS of the destination IP address, or `FAIL` otherwise. An IP routing agent infers the status of an *AS Hop* from the result of *AS Hop Tests*, and can infer the status of an *AS Path* from *AS Path Tests* and *AS Hop Tests* along the AS path from the source to the destination. It can then infer the status of the *IP Routing* component. In this prototype implementation, this test is performed using the Scriptroute Rockettrace tool [81] and destination ASes are looked up using whob.[3] Note that an agent may not always be able to test the status of an AS hop if the agent is not along the AS path from the user to the web server or if a failure occurs before reaching the source AS.

8.4 Routing of diagnostic information

Fault diagnosis in this implementation requires the exchange of diagnostic information among multiple agents. In order to compute aggregate statistics about web servers, server history agents subscribe to notifications of *HTTP Connections*. Learning agents subscribe to information about diagnostic results. In the process of diagnosis, notifications of diagnostic information flow from user agents to a regional agent and then to server history agents and a learning agent. Regional agents can then request the observations, beliefs, and knowledge generated by the web server history and learning agents. User agents may then request dependency knowledge from regional agents. Regional agents aggregate requests from user agents, request additional diagnostic information from specialist agents, and finally respond to requests. In this section I describe the routing and aggregation patterns of diagnostic information in my prototype implementation.

Notification subscriptions allow certain agents to aggregate certain types of diagnostic information. In order to learn dependency knowledge for diagnosing failures, knowledge agents collect observations about *HTTP Connections* and their associated *Local Network*

[3] http://pwhois.org/lft/

and *HTTP Server* statistics. Rather than collecting server connection history observations directly from users, which does not scale well with the number of users, the learning agent requests aggregate data from server history agents and subscribes to notifications of diagnostic results from regional agents. Note that a learning agent can perform this data collection periodically and do learning offline to reduce communication cost.

User agents make observations about *HTTP Connections*, *Outbound Conn Tests*, and *Firefox Error Tests*. User agents periodically send these observations to regional agents, which then forward on the observations to server history diagnostic agents, which then compute the `consecFailuresToServer` for each destination MD5 hash. All *HTTP Connections* with the same destination MD5 hash get sent to the same server history agent. User agents also periodically send aggregate statistics on `consecFailuresFromUser` and the results of its diagnostic tests to a regional diagnostic agent.

Here is an example of a *HTTP Connection* observation made by a user agent:

```
<notification
  xmlns:core="http://capri.csail.mit.edu/2006/capri/core#"
  xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
  version="0.3"
  time="12345"
  clientVer="0.30"
  notificationID="10">
<body>
<observation>
  <component id="1">
    <class>
      http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
    </class>
    <com:destHash>+0THLqaZnxk9I8bU5ZgDGA==</com:destHash>
    <com:connTime>123</com:connTime>
    <core:status>OK</core:status>
    <com:elapsedTime>340</com:elapsedTime>
  </component>
</observation>
</body>
</notification>
```

Next, the learning agent collects `consecFailuresToServer` statistics from server history agents. The learning agent labels past failures, learns dependency knowledge, and then communicates this dependency knowledge to regional diagnostic agents, which then communicate this knowledge to user agents.

Diagnostic requests flow from users to regional diagnostic agents. Regional diagnostic agents may then request additional information from server history agents, AS path test agents, or DNS lookup test agents. A regional agent can reduce the cost of diagnosis by incorporating cached information in its component information base relevant for diagnosis. A regional agent may also request information from other agents to maximize the expected future value of that information. This process is a form of request aggregation in that

it enables an agent to diagnose multiple failures using fewer tests and requests than is required to diagnose each failure individually.

Regional agents may communicate additional observations about *HTTP Server* components to user agents in their diagnostic responses to reduce the cost of diagnosing future failures. For example, if a web server fails, the server agent for that web server will notice that multiple users cannot connect to that web server and increase the `consecFailuresToServer` statistic for that web server. A regional agent diagnosing the failure requests the `consecFailuresToServer` information from the server agent and uses it to diagnose the *HTTP Connection* failures from users. The regional agent also communicates the `consecFailuresToServer` statistics to user agents so that the user agents can diagnose future failures in connecting to the failed web server.

The web server history agent collects and aggregates HTTP connection history data from user agents to produce `consecFailuresToServer` observations, the number of consecutive unique-user failures to a destination web server IP address. The web server history agent combines this aggregate data from users with its dependency knowledge to compute a belief about the status of the destination web server.

For additional scalability and to balance the load of diagnosis, in this experiment there are four web server history agents distributed throughout Planetlab, each responsible for a different set of web server IP addresses. Each web server history agent subscribes to *HTTP Connection* observations for a different set of web servers. This demonstrates the ability of CAPRI to distribute dependency knowledge and diagnostic capabilities among multiple agents.

8.5 Diagnosis procedure

The procedure for diagnosis described in Chapter 7 allows agents without any previous knowledge of these types of diagnostic information to automatically learn the meaning of these component and test classes from the component ontology and diagnose failures using dependency knowledge collected from other agents. This allows regional diagnostic agents to automatically construct failure dependency graphs from observations of connection failures from users and dependency knowledge from server history agents. Agents in this implementation follow the following procedures for fault diagnosis.

8.5.1 User agent

When a user agent first starts (i.e. a user opens their Firefox web browser), it checks its service table to determine the regional agents it can use. If its service table has expired, it requests new services from the agent directory. A user agent expires a service 24 hours after receiving it from the agent directory.

When a user agent detects an HTTP connection failure, it collects data about the Firefox error code associated with the failure and creates the corresponding observations of the *HTTP Connection* and *Firefox Error Test*. A failure is defined as an HTTP request that results in a Firefox error code corresponding to a network error. In addition, because

many network failures manifest themselves as slow-loading web pages, failures also include HTTP requests that the user cancels (i.e. presses the "Stop" button in their browser or navigates away from a page) after 15 seconds without a response from the web server.

When a failure occurs, the user agent also computes the `consecFailuresFromUser` statistic for the *Local Network* component corresponding to the user's local network. The user agent then checks its component information base to determine whether it has any information about the *HTTP Server* and *Local Network* components for this *HTTP Connection*. If so, the user agent fills in the properties of the *HTTP Server* and *Local Network* components using information from its component information base. The user agent then constructs a failure dependency graph containing the failed *HTTP Connection*, the *Firefox Error Test*, the *Local Network*, a *DNS Lookup*, and the *HTTP Server*. All user agents have dependency knowledge stating that the status of an *HTTP Connection* depends on the *Local Network* `status`, *DNS Lookup* `status`, *IP Routing* `status`, and *HTTP Server* `status`. If the user agent has probabilistic dependency knowledge for inferring the probability of *HTTP Server* failure and *Local Network* failure from `consecFailuresToServer` and `consecFailuresFromUser`, then it adds the corresponding edges to the failure dependency graph.

The user agent then performs probabilistic Bayesian inference using its failure dependency graph to compute the likelihood of failure for the *Local Network*, the *DNS Lookup*, *IP Routing*, and *HTTP Server* components in its failure dependency graph. If its confidence in its diagnosis is insufficient, the agent then conducts an *Outbound Conn Test* and performs diagnostic inference again. If it still does not have sufficient knowledge to diagnose the failure, or if its confidence in its diagnosis is too low, then the user agent requests diagnosis from the regional diagnostic agent. The user agent then constructs a failure report containing the information in its failure dependency graph and transmits a diagnostic request containing this failure report to the regional diagnostic agent.

Every five minutes, a user agent sends a batch of HTTP connection history notifications and diagnosis history notifications to a regional agent.

8.5.2 Regional agent

Regional agents act as dispatchers and aggregation points. Using the general, dynamic procedure for processing messages described in Chapter 7, they identify possible similarities among diagnostic requests and decide which agent to contact next based on the value of the information each agent provides. When a regional diagnostic agent receives a diagnostic request from a user agent, it takes the component graph in the request and adds any relevant information it has in its component information base, such as the `consecFailuresToServer` for the destination host or the results of other previous tests. A regional agent also computes the AS path for the IP route from the user to the destination web server in order to identify potentially similar failures. The regional diagnostic agent requests new dependency knowledge if necessary. It then performs probabilistic inference to infer the cause of failure. If the regional agent's confidence in its diagnosis exceeds a threshold, then it returns a diagnosis to the user agent. Otherwise, it computes possible next actions, requests additional tests from specialist agents as necessary, and repeats its diagnostic inference.

A regional agent uses information in its component information base to identify similar

failures. When a regional agent receives a diagnostic request, it has three options for obtaining additional information about the components involved in the failure: it may request web server status beliefs from a server history agent, DNS lookup status beliefs from a DNS lookup verification agent, or IP routing status beliefs from an AS path test agent. In order to control the cost of diagnosis, a regional agent selects the action with greatest utility, where utility is the expected confidence of diagnosis using the information produced by the action divided by the cost of that action. The procedure for probabilistic inference provided by CAPRI can automatically take into account the value of information of various tests based on available evidence and dependency knowledge.

Note that a regional agent does not need the ability to perform any diagnostic tests; it simply requests information it needs from other agents. Using the dependency knowledge, beliefs, and observations it obtains, a regional agent can diagnose failures in a general way. This procedure makes it easy to add additional specialist agents, component classes, and diagnostic tests.

8.5.3 Web server history test agent

A web server history test agent is one of four types of specialist agents in this prototype. The server history agent is a type of passive diagnostic agent that does not perform any active probes or diagnostic tests, but rather simply collects observations from other agents in order to produce aggregate statistics useful for diagnosis. A server history agent collects observations of *HTTP Connections* from many users to a single *HTTP Server* IP address. It then uses these observations to compute the number of consecutive users who cannot connect to the *HTTP Server*. A server history agent assigns this value to the `consecFailuresToServer` property of the corresponding *HTTP Server* component. A server history uses this information to infer a belief in the status of the HTTP server. Server history agents respond to belief requests for HTTP server status given the IP address of a server as input.

To demonstrate the possibility of distributed diagnosis using multiple server history agents, this prototype contains four server history agents, each responsible for a different set of destination IP addresses. Web server history test agents specify input restrictions in their service advertisements to achieve this distribution of responsibility. A regional agent dynamically determines the appropriate server history agent to use by consulting the service descriptions in its service table.

8.5.4 DNS lookup test agent

A DNS lookup test agent infers beliefs about the status of a *DNS Lookup* component given a hostname and the IP addresses returned by a lookup. In this prototype implementation, the DNS lookup verification agent performs another DNS lookup and compares the IP addresses it obtains with the IP addresses obtained by the user when the HTTP failure occurred. This test can detect several types of DNS lookup problems. For instance, suppose that the user was not able to obtain an IP address for the destination hostname. If the DNS lookup verification agent is able to get an IP address, it may indicate that the user's DNS servers have failed. Alternatively, if the IP addresses obtained by the user and the

DNS agent differ, it may indicate that the user's DNS server has incorrect or out of date information. A DNS lookup verification agent can respond to belief requests for *DNS Lookup* `status` given a hostname and a set of IP addresses as input.

This prototype implementation contains three DNS lookup verification agents. Each agent can diagnose any hostname. A regional agent selects a DNS lookup test agent by consulting the service descriptions in its service table.

8.5.5 AS path test agent

AS path test agents infer beliefs about the status of an *IP Routing* component by performing Rockettrace tests. A Rockettrace test provides information about the status of IP routing across different ASes.

An AS path test agent responds to belief requests for *IP Routing* component status. In a belief response, an AS path test agent also includes beliefs about the status of the AS hops along the IP path in order to assist regional agents with aggregation.

Since the ability of an AS path test agent to test a given IP route depends on its location in the Internet network topology, the AS path test agent specifies an input restriction in its service description to only accept requests for IP routes for which the source IP is within the same AS as the AS path test agent.

Note that many existing traceroute tools exist today, and many services exist to perform traceroutes [63].[4] CAPRI allows one to wrap such existing tools and services so that other agents can automatically request such data and incorporate it for diagnosing failures.

8.5.6 Stats agent

The four stats agents in this prototype demonstrate the ability to aggregate observations of HTTP connections from multiple users to a destination hostname and produce observations of aggregate statistics. To distribute responsibility among the four stats agents, each stats agent specifies an input restriction on the set of web server hostname hashes for which it can provide information.

8.5.7 Knowledge agent

Agents diagnose failures using probabilistic inference and Bayesian networks. To perform such probabilistic inference, agents need probabilistic dependency knowledge. Some of this dependency knowledge may come from expert humans, but it may also be learned automatically by agents. In this prototype, I demonstrate how a knowledge agent can learn dependency knowledge from observations of past failures and communicate this knowledge to other agents.

A knowledge agent creates new dependency knowledge by reconstructing failure dependency graphs for past failures, labeling each diagnosis with the true cause of failure, and then learning the probabilistic dependencies using a Bayesian learning algorithm. In my prototype implementation, the knowledge agent collects evidence about failures and

[4] http://www.bgp4.net/wiki/doku.php?id=tools:ipv4_traceroute

diagnostic tests using a notification subscription. Once a day, the knowledge agent also requests `totalConsecFailuresToServer` observations from web server history agents for each diagnosis notification it receives.

Next the knowledge agent labels each diagnosis with the true cause of failure. In this prototype implementation, it is not always clear what the true cause of failure is in each case, however. Therefore in this implementation a knowledge agent uses the set of rules in Algorithm 1 for labeling the cause of failure. I chose the rules below because they appear to classify failures fairly accurately for most of the real-world network failures I examined. The number of `totalConsecFailuresFromUser` for a failure represents the total number of consecutive failures to unique destinations both before and after the failure in question, whereas `consecFailuresFromUser` only counts the number of consecutive failures before the failure in question. For example, if a user cannot connect to three consecutive different destinations, then the number of total consecutive failures from the user is three for all three failures. Similarly, `totalConsecFailuresToServer` represents the total number of consecutive failures from unique users both before and after the failure in question.

Algorithm 1 The procedure for labeling failures using a posteriori information

if `totalConsecFailuresFromUser` ≥ 3 **then**
  Label it a local network failure.
else if `totalConsecFailuresToServer` ≥ 2 **then**
  Label it a web server failure.
else if *Outbound Conn Test* `probeResult` $=$ `FAIL` **then**
  Label it a local network failure.
else if *Firefox Error Test* `ffoxErrorCode` $= 30$ (server not found) **then**
  Label it a DNS lookup failure.
else if *AS Path Test* `asPathTestResult` $=$ `FAIL` **then**
  Label it an IP routing failure.
else if *Firefox Error Test* `ffoxErrorCode` $= 13$ (connection refused) **then**
  Label it a web server failure.
else if *AS Path Test* `asPathTestResult` $=$ `OK` **then**
  Label it a web server failure.
else if *Firefox Error Test* `ffoxErrorCode` $= 14$ (connection timed out) **then**
  Label it an IP routing failure.
else
  Label it an unknown failure.
end if

Once a learning agent learns dependency knowledge, it communicates this knowledge to regional and specialist agents. Because the distribution of failures in the Internet changes frequently, knowledge learned at one point in time may differ from knowledge learned at another time. Therefore it is important to continually learn and communicate new dependency knowledge. The optimal frequency with which a learning agent collects observations and communicates knowledge depends on the rate at which dependency knowledge changes, the cost of collecting observations, and the tradeoffs one is willing to make between accuracy and cost.

Though it is possible to have multiple learning agents to distribute the communication and component cost of learning, having fewer learning agents improves accuracy because then each agent has a more complete set of observations from which to learn.

8.6 Extensibility

The strength of the CAPRI architecture is that it allows one to introduce new diagnostic agents and diagnostic tests, and diagnose failures using different patterns of communication. This implementation can be extended to support new types of agents and information in several ways.

Initially, this implementation has four types of specialist agents: web server history agents, DNS lookup test agents, AS path test agents, and stats agents. The strength of the CAPRI architecture is that it allows one to add new agents or enhance existing agents with new diagnostic tests. For example, instead of simply performing another DNS lookup, a DNS lookup agent might also attempt to validate DNS server configuration by checking for inconsistencies [29]. An AS path agent might use other tools in addition to traceroute to test AS hops. A web server diagnosis agent may also perform active probes or measure statistics such as latency and failure rate.

To demonstrate extensibility in my experiments, I add a new specialist agent to the system that uses the CoDNS service [69] to provide beliefs about *DNS Lookup* `status`. Chapter 9 describes the results of these experiments.

In addition, one can imagine other types of specialist agents as well. For example, a web browser specialist agent might be able to determine whether a failure was due to a software bug. An IP link specialist agent might be able to identify individual IP link failures caused by router or switch failures. To add a new specialist agent, we need to make three modifications to this implementation:

1. Add new component class and property definitions to the component ontology if necessary. For example, to diagnose DNS servers one must define a *DNS Server* class and add a component relationship property to the *DNS Lookup* class relating a *DNS Lookup* with a *DNS Server*.

2. Advertise the new capabilities of the agent to the agent directory. For example, a DNS specialist agent might provide information about the probability of *DNS Lookup* and *DNS Server* failure given a *DNS Server* IP address and a hostname.

3. Optionally provide new dependency knowledge for any newly defined properties by advertising a knowledge service. For example, a DNS server agent might provide knowledge of the conditional probability of *DNS Lookup* `status` given the `status` of the corresponding *DNS Server*. Note that a specialist agent need not reveal this knowledge; it may choose to keep this knowledge private if it wishes. If the agent does reveal this knowledge, however, then other agents can infer the `status` of a *DNS Lookup* from observations of *DNS Server* `status` without requesting diagnosis from the specialist agent.

Also, note that there are several ways in which a specialist agent might operate. For example, a server history agent might only provide observations of consecutive failures to a server without having any knowledge of the probabilistic dependencies between this statistic and the status of an HTTP server. Then another agent with the proper knowledge can perform inference of HTTP server status. Alternatively, the agent might provide only the beliefs without providing the evidence. For this to work, a requesting agent must provide the necessary likelihood or beliefs for belief propagation in order to get an accurate result, possibly involving more communication overhead. A third possibility, and the one I implement here, is for the specialist agent to provide both the knowledge and the evidence so that the requesting agent can aggregate more effectively with fewer messages and without prior knowledge of the methods that the specialist agent uses for diagnosis.

Another way to extend this prototype is to introduce new diagnostic test classes and new properties of previously defined component classes. For example, in addition to `consecFailuresToServer`, the server history agent might also provide statistics on the average rate of failure to particular destinations. To add such new diagnostic tests and properties, one must define the new property, create an agent that advertises the capability to provide observations of the new property, and optionally provide new diagnostic knowledge.

Another possibility is to use existing agents in different ways. For instance, in this implementation the AS path test agents are only used to run traceroutes from the user to the server. It may also be useful to use AS path test agents to attempt to run traceroutes from the server to the user, for example.

8.7 Controlling cost

The diagnostic procedure above controls costs of communicating diagnostic observations, knowledge, and requests by aggregating multiple similar failures and distributing requests across multiple agents.

HTTP Connection information from all users is distributed among multiple server history agents based on the destination of the *HTTP Connections*. Thus a single server history agent only needs to handle a fraction of the total amount of connection history data. This architecture allows us to support additional users by adding more server history agents and redistributing the set of destination servers among the server history agents.

Each user agent requests diagnosis from a particular regional agent. Thus to support additional users while limiting the number of user agents for which a single regional agent diagnoses failures, we can add additional regional agents and assign the new users to these additional regional agents. Regional agents also act as aggregation points for requesting observations from server history agents and dependency knowledge from learning agents, limiting the load of diagnostic requests on server history and learning agents. In addition, aggregating multiple diagnostic requests at a single regional agent allows a regional agent to diagnose multiple *HTTP Connection* failures caused by the failure of a single *HTTP Server*. This architecture also means that user agents do not need to know what specialist agents exist or which ones to contact to diagnose a failure.

Note that adding new regional agents involves a tradeoff. Increasing the number of regional agents reduces the number of requests each one needs to handle, but it also reduces

opportunities to use cached information to diagnose multiple similar failures. The optimal number of regional agents depends on many factors, including the frequency of failures, the burstiness of requests, the rate at which regional agents can process requests, the probability that cached information collected during the diagnosis of one failure can be used for diagnosing future failures, and the assignment of user agents to regional agents.

In this implementation, aggregation of requests occurs at both user agents as well as at regional agents. User agents can cache information about the results of outbound connectivity probes and the number of consecutive failures to particular destinations. A user agent may then use this information to diagnose future failures with fewer requests.

In order to learn an accurate dependency model for *HTTP Connection* failures, a learning agent must collect observations for all *HTTP Connection* failures. Therefore it subscribes to notifications of historical observations of *HTTP Connection* failures and their associated diagnostic tests and dependent components from regional agents and requests aggregate data from server history agents. Since dependency knowledge changes relatively slowly compared to statistics such as `consecFailuresToServer`, a learning agent does not need to receive immediate notifications of every failure and can just collect batched observations of *HTTP Connections* periodically.

8.8 Diagnosis with incomplete information

Diagnostic information and capabilities are distributed over multiple diagnostic agents in this prototype implementation. Diagnostic agents use probabilistic inference to compute the probability of the most likely cause of failure and the confidence of their diagnosis with possibly incomplete information. In addition, agents can propagate useful observations and knowledge to other agents so that they can use this information to diagnose future failures. Probabilistic inference combined with propagation of information allows diagnostic agents to diagnose failures even when other diagnostic agents fail or are unreachable. For example, if a user agent cannot contact a regional agent for diagnosis but has dependency knowledge about how to infer the most likely cause of failure from `consecFailuresFromUser`, it can still provide an approximate diagnosis without knowing the results of other tests. Similarly, even if a regional agent cannot obtain up-to-date `consecFailuresToServer` observations from a server history agent, the regional agent may still use cached observations of `consecFailuresToServer` in its diagnosis.

The ability to perform diagnosis with incomplete information also provides robustness to network failures and agent failures. For example, even though a user without network connectivity may not be able to connect to their regional agent, the user agent can use outbound connectivity tests and observations of consecutive failures to diagnose a failure with incomplete information.

Diagnosis with incomplete information also gives users the option to trade off privacy and accuracy. In my prototype implementation, knowing the identity of the destination web server can greatly improve the accuracy of HTTP connection failure diagnosis, but some users may not wish to reveal such personal information. In general, the more information that a diagnostic agent has, the more options it has for diagnosis and the more accurate its diagnosis. CAPRI allows agents to trade off privacy and accuracy by allowing a diagnosis

requester to provide as little or as much information as they wish. For example, rather than revealing the full URI of the destinations that users wash to reach, a user agent might only reveal an MD5 hash of the destination hostname. This potentially reduces the accuracy of diagnosis because although it still provides server diagnostic agents with enough information to determine if multiple users experience failures connecting to the same destination hostname, it prevents an agent from determining how many failures have occurred to the destination IP address or whether the DNS lookup completed successfully. On the other hand, it provides additional privacy because the requester does not reveal the actual identity of the destination host. Even though hashing the hostname does not provide very strong security because an agent may attempt a dictionary attack to find hash collisions, it does illustrate how a diagnosis requester may make tradeoffs between privacy and accuracy.

8.9 Agent implementation details

All regional and specialist agents are deployed on Planetlab nodes. The service directory and knowledge agent run on a machine at MIT. User agents in this implementation operate as Firefox extensions inside web browsers on end user machines. Implementing user agents as Firefox extensions has a number of advantages. Firstly, Firefox supports multiple platforms, including Windows, MacOS, and Linux. Secondly, Firefox has an active community of extension developers. Thirdly, the Firefox addons web site[5] provides a convenient distribution channel to reach a large number of users around the world. For platform independence, I implemented user agents using Javascript. User agents do not advertise any services; they only issue notifications and diagnosis requests.

All other agents are implemented as Python 2.4 scripts running as Apache mod_python handlers interfacing with a MySQL database backend. There are several advantages to implementing agents using Apache, Python, and MySQL.

Firstly, Apache is multithreaded, allowing agents to handle multiple requests simultaneously. Secondly, Apache can demultiplex requests to multiple diagnostic agents operating as HTTP request handlers on the same web server. Thirdly, Apache provides a common logging infrastructure. Fourthly, Apache provides tools for monitoring HTTP server status to assist in debugging. A disadvantage of Apache is the additional memory and CPU overhead of running a web server.

Python has several advantages as well. Firstly, as a high-level scripting language that provides many built-in features and modules, it allows for much more rapid development than C or Java. Secondly, Python is a popular and easy to understand language, which makes it easier for others to build their own agents using the code I developed. A disadvantage of Python is that it is an interpreted language, and is slower and uses more memory than a natively compiled program written in C or C++. This disadvantage is somewhat mitigated by the fact that Python does support modules written in C. I decided to accept this tradeoff because my experiments primarily evaluate the functionality of CAPRI, and not the speed of diagnosis.

I choose to use a MySQL database rather than memory or disk files for storing data in

[5] https://addons.mozilla.org/en-US/firefox

this implementation for several reasons. Firstly, a database provides atomicity and consistency to support multithreaded operation. Secondly, a database provides a standard SQL interface for querying and aggregating information agents collect. Each Python agent maintains its component information base, knowledge base, service table, and component class and property definitions in database tables. In addition, web server history agents, stats agents, and knowledge agents record and aggregate notifications they receive using connection history and diagnosis history tables.

Agents send and receive service advertisements using a centralized agent directory server located at MIT. All agents know the URI of the service directory. In order to discover newly available services, Python agents reload their list of available services from the agent directory approximately once an hour. To take advantage of new regional agents that become available, user agents reload available services approximately once a day. Regional and specialist agents use requester AS restrictions and costs to preferentially handle requests from requesters in the same AS. Thus users who are in the same AS as a regional agent send diagnostic requests and notifications to that regional agent. Similarly, regional agents request beliefs from specialist agents within their AS if available.

To support the creation of new diagnostic agents, most of the common functionality of CAPRI agents such parsing messages, maintaining a component ontology, managing a service table, constructing failure dependency graphs, and performing probabilistic inference is contained in a Python module. Python agents use the OpenBayes[6] module to perform probabilistic inference and use the PyXML module[7] to parse XML messages. Knowledge agents use the SMILE reasoning engine to learn probabilistic dependencies.[8] Creating a new diagnostic agent simply involves extending the default diagnostic agent implementation with the desired capabilities and placing new component class and property definitions and service descriptions in the appropriate locations.

Below are tables of the notification subscriptions and the services offered by diagnostic agents in my prototype implementation. Input and requester restrictions are not shown here. In addition, most services also have a corresponding "local" version with lower cost for those requesters in the same AS as the agent. Appendix B contains the full service descriptions.

[6]http://www.openbayes.org/

[7]http://pyxml.sourceforge.net/

[8]SMILE is available from the Decision Systems Laboratory, University of Pittsburgh (http://dsl.sis.pitt.edu)

Service ID	Input	Output	Cost
diag:http	*HTTP_Connection* status dnsLookup.hostname localNet.srcIP ipRouting.destIP	**Explanations** localNet.status httpServer.status dnsLookup.status ipRouting.status	1000
obs:stats	*HTTP_Connection* destHash (required)	**Observation** users avgLatency lastSuccess lastFailure recentStatusDist	1000
knowledge:http		**Knowledge** Local_Network.status Local_Network.consecFailuresFromUser HTTP_Connection.status DNS_Lookup.status IP_Routing.status HTTP_Server.status HTTP_Server.consecFailuresToServer Outbound_Conn_Test.probeResult Firefox_Error_Test.ffoxErrorCode	100

Table 8.1: Regional agent services

Agent/Service ID	Input	Output	Cost
Web Server History Agent bel:webserver.status	*HTTP_Server* ipAddr (required) connTime (required) hostHash httpConn httpConn.status httpConn.destHash httpConn.destIP httpConn.srcIP httpConn.connTime httpConn.elapsedTime httpConn.ffoxErrorTest httpConn.ffoxErrorTest.- ffoxErrorCode	**Belief** status **Observation** consecFailuresToServer **Knowledge** HTTP_Server.status	1000
Web Server History Agent obs:webserver.cfts	*HTTP_Server* ipAddr (required) connTime (required)	**Observation** consecFailuresToServer totalConsecFailuresToServer	1000
DNS Lookup Agent bel:dnslookup.status	*DNS_Lookup* hostname (required)	**Belief** status **Observation** verifyDNSLookupTest.- dnsLookupResult **Knowledge** DNS_Lookup.status	10000
AS Path Agent bel:iprouting.status	*IP_Routing* srcIP (required) destIP (required)	**Belief** status **Observation** asPathTest.asPathTestResult **Knowledge** IP_Routing.status	100000
Stats Agent obs:stats	*HTTP_Connection* destHash (required)	**Observation** users avgLatency lastSuccess lastFailure recentStatusDist	100

Table 8.2: Specialist agent services

Service ID	Input	Output	Cost
knowledge:http(learned)		**Knowledge** Local_Network.status Local_Network.consecFailuresFromUser HTTP_Connection.status DNS_Lookup.status IP_Routing.status HTTP_Server.status HTTP_Server.consecFailuresToServer Outbound_Conn_Test.probeResult Firefox_Error_Test.ffoxErrorCode Verify_DNS_Lookup_Test.dnsLookupResult AS_Path.status AS_Path_Test.asPathTestResult AS_Hop.status AS_Hop_Test.asHopTestResult	1000
knowledge:specialist		**Knowledge** Local_Network.status Local_Network.consecFailuresFromUser DNS_Lookup.status IP_Routing.status HTTP_Server.status HTTP_Server.consecFailuresToServer Outbound_Conn_Test.probeResult Verify_DNS_Lookup_Test.dnsLookupResult AS_Path.status AS_Path_Test.asPathTestResult AS_Hop.status AS_Hop_Test.asHopTestResult	1000

Table 8.3: Knowledge agent services

Service ID	Input
notify:connHist	*HTTP_Connection* status destHash srcIP destIP connTime elapsedTime ffoxErrorTest ffoxErrorTest.ffoxErrorCode
notify:diagHist	*HTTP_Connection* localNet localNet.status localNet.ipAddr localNet.consecFailuresFromUser localNet.totalConsecFailuresFromUser httpServer httpServer.status httpServer.consecFailuresToServer dnsLookup dnsLookup.status dnsLookup.hostname ipRouting ipRouting.status ipRouting.srcIP ipRouting.destIP outboundConnTest outboundConnTest.probeResult outboundConnTest.probeURI ffoxErrorTest ffoxErrorTest.ffoxErrorCode

Table 8.4: Regional agent notification subscriptions

Agent/Service ID	Input
Stats Agent notify:connHist(destHash)	*HTTP_Connection* status destHash (required) srcIP destIP connTime elapsedTime ffoxErrorTest ffoxErrorTest.ffoxErrorCode
Web Server History Agent notify:connHist(destIP)	*HTTP_Connection* status destIP (required) srcIP connTime elapsedTime ffoxErrorTest ffoxErrorTest.ffoxErrorCode

Table 8.5: Specialist agent notification subscriptions

Service ID	Input
notify:diagHist	*HTTP_Connection* localNet (required) localNet.status (required) localNet.ipAddr (required) localNet.consecFailuresFromUser (required) localNet.totalConsecFailuresFromUser httpServer (required) httpServer.status httpServer.consecFailuresToServer dnsLookup (required) dnsLookup.status dnsLookup.hostname ipRouting (required) ipRouting.status ipRouting.srcIP ipRouting.destIP outboundConnTest (required) outboundConnTest.probeResult outboundConnTest.probeURI ffoxErrorTest (required) ffoxErrorTest.ffoxErrorCode

Table 8.6: Knowledge agent notification subscriptions

Chapter 9

Experimental Results

The purpose of the CAPRI architecture is to provide a common framework for distributed diagnosis among heterogeneous diagnostic agents. This chapter evaluates the effectiveness of distributed diagnosis using the diagnostic agents in the prototype implementation described in Chapter 8. I demonstrate that multiple heterogeneous diagnostic agents with different capabilities and located in different parts of the network can use CAPRI to effectively diagnose real-world HTTP connection failures in a distributed manner in real time. Unlike previous research in distributed diagnosis [44, 67, 98], agents in CAPRI can incorporate information from multiple types of diagnostic tests, including both active and passive measurements, and can deal with incomplete information caused by network failures. In addition, I demonstrate the ability to add new diagnostic agents to the system and show that existing diagnostic agents can take advantage of new dependency knowledge and new diagnostic agents to improve the accuracy and cost of diagnosis. This chapter also illustrates the effectiveness of aggregation and caching to reduce the cost of diagnosing multiple similar failures for scalability. Unlike previous research that only considers the probing costs of single diagnoses [76], this experiment shows how agents can manage both the probing and communication costs of multiple, repeated diagnoses.

This chapter presents experimental results demonstrating the advantages of the CAPRI architecture. I show that regional agents dynamically select diagnostic actions without domain-specific knowledge. Heterogeneous diagnostic agents use belief propagation to diagnose failures in a distributed way even when no single agent has enough information to adequately diagnose a failure. I also find that caching and probabilistic inference reduces the number of diagnostic requests sent to specialists and to regional agents. Also, I show that adding learned dependency knowledge improves the accuracy of diagnosis while reducing cost. In addition, adding a new type of diagnostic agent can reduce the response time of certain requests. I also show that the confidence metric produced in a diagnosis can be used to predict the expected accuracy of a diagnosis if the dependency knowledge used for diagnosis is correct.

The experimental evaluation described in this thesis differs from previous work in distributed Internet fault diagnosis in that it evaluates the performance of real-time, online distributed diagnosis for real-world failures. Previous researchers generally focus on offline diagnosis or diagnosis of simulated failures in simulated networks. Evaluation using real-world failures can provide better insight into the challenges of diagnosis under real-

world conditions, in which network failures can interfere with diagnostic communication and diagnostic tests may produce noisy or inaccurate results. The data for the experimental results shown below were collected over a period of over one month, from March 6 to April 10, 2007. In this experiment, diagnostic agents collect information about a total of approximately 500,000 HTTP connections per day, and diagnose approximately 3,000 HTTP connection failures per day.

9.1 Distributed diagnosis

Chapter 8 describes the range of diagnostic agents that I implemented and deployed. Figure 9-1 illustrates the knowledge that agents use to diagnose failures in my implementation. Each type of diagnostic agent knows a different subset of this dependency knowledge. The CAPRI architecture enables distributed agents to discover one another and communicate observations, beliefs, and knowledge to perform distributed fault diagnosis.

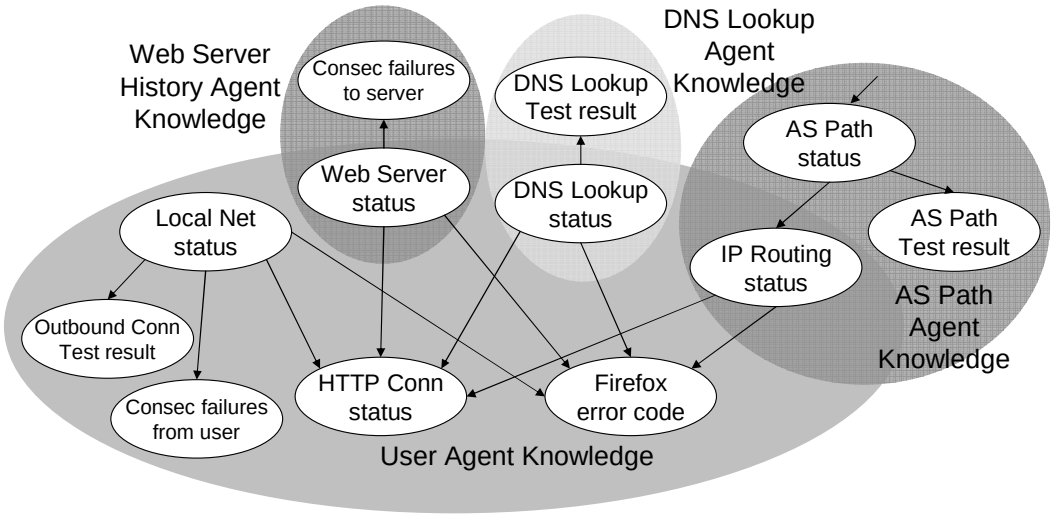

Figure 9-1: Agents with different capabilities and knowledge exchange observations and beliefs to diagnose failures.

9.2 Dynamic action selection

Agents in CAPRI diagnose failures dynamically based on the information in a request, service descriptions, and dependency knowledge. This general, dynamic message processing procedure allows agents to automatically take into account new dependency knowledge and new services when they become available. Figure 9-2 illustrates diagnostic requests that regional agents receive and the requests that they make to specialist agents for 27,641

diagnostic requests from March 6 to March 21, 2007. Table 9.1 presents this data in tabular form. Given a Firefox error code and a combination of specialist agents, the table indicates the number of diagnoses for which a regional agent contacted all of those specialist agents to diagnose a failure of that type. The figure shows that regional agents are able to dynamically decide what actions are appropriate in a non-domain-specific way, based only on the information in the request, service descriptions, and dependency knowledge.

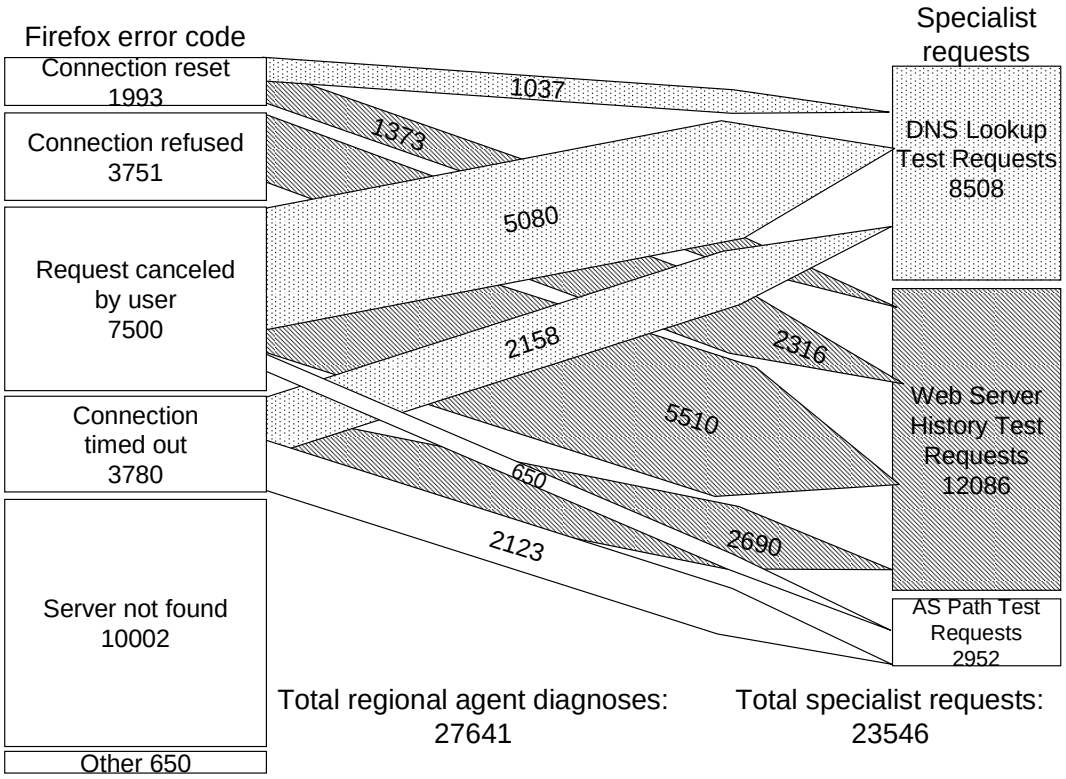

Figure 9-2: Regional agents dynamically select specialist agents to contact.

Diagnostic requests at the left of the figure are categorized by the error code reported by the Firefox web browser. The arrows indicate the total number of diagnostic requests regional agents made to each type of specialist agent. Note that for clarity, arrows with fewer than 100 requests are not shown. Certain types of failures are easier to diagnose than others. For example, regional agents respond to diagnostic requests for which the Firefox error code is “server not found” using only probabilistic dependency knowledge without any additional diagnostic tests. Such failures are with very high probability DNS Lookup failures. This capability to diagnose failures with high accuracy with incomplete information in a general way is one important advantage of probabilistic diagnosis. For other types of failures, such as “request canceled by user”, a regional agent may need to request additional tests from multiple other specialist agents.

This figure also shows the ability of agents in CAPRI to take into account the cost of

Firefox error code	Diagnoses using indicated specialists W = Web Server Test D = DNS Lookup Test A = AS Path Test none	W	D	WD	A	WA	DA	WDA	Total
Canceled by user	1020	1265	866	3717	56	79	48	449	7500
Connection refused	1378	2250	21	26	27	2	9	38	3751
Connection timed out	665	506	64	422	175	276	186	1486	3780
Connection reset	474	467	134	846	6	9	6	51	1993
Server not found	9997	1	2	0	0	2	0	0	10002
Other	383	80	25	80	5	10	8	24	615
Total	13917	4569	1112	5091	269	378	257	2048	27641

Table 9.1: Distribution of specialist agent requests from regional agents by Firefox error code

different tests when selecting actions. In my prototype implementation, some services have greater costs than others. Web server history tests have the lowest cost, followed by verify DNS lookup tests, while AS path tests have the greatest cost. For this reason, using the dynamic procedure for action selection provided by CAPRI, agents will typically request additional diagnostic tests useful for diagnosis in order of increasing cost.

This figure also illustrates the ability of agents to automatically decide what actions are possible based on the inputs specified in service descriptions. In certain cases, agents do not have the necessary inputs to request certain diagnostic tests. For example, when a "server not found" error occurs, the user agent does not have an IP address for the destination web server and so cannot conduct a web server history test or an AS path test.

CAPRI also allows agents to preferentially handle requests from nearby agents by advertising lower costs. Three out of the 14 regional agents in my experiments can request DNS lookup tests from specialist agents located in the same AS. Because DNS lookup specialist agents advertise a lower cost of diagnosis to regional agents within the same AS (9000 instead of 10000), regional agents in the same AS as a DNS specialist agent request DNS lookup tests from specialist agents within the same AS. In this experiment I found that 2801 out of 8508 (33%) of DNS lookup requests are handled by DNS specialist agents in the same AS as the regional agent.

The procedure that agents use action selection also enables an agent to automatically work around failures in other agents simply by considering the value of the information actions provide. For example, at one point in my experiments a DNS lookup agent at the University of Oregon became unreachable due to a Planetlab software upgrade. Because multiple DNS lookup specialists offer the same services, a regional agent has multiple DNS lookup test request actions in its list of possible next actions, all with the same value. A regional agent that fails to connect to the University of Oregon DNS lookup agent selects another DNS lookup test action with the same value if available. Once a regional agent successfully obtains DNS lookup test information from a specialist, however, the value of all other DNS lookup test actions becomes zero because they do not provide any new information.

These experimental results illustrate some of the benefits of dynamic message processing and action selection based on action value and cost. The action selection procedure that CAPRI agents use automatically distinguishes between possible and impossible actions based on the information in an agent's component graph, and can take into account the different costs of actions to preferentially request lower cost tests. This procedure allows an agent to automatically work around failures in other agents and estimate the expected gain in accuracy of available diagnostic actions without domain-specific knowledge.

9.3 Probabilistic diagnosis

This thesis proposes the use of probabilistic models for representing diagnostic information and dependencies and making inferences. This section presents experimental results demonstrating several benefits of probabilistic diagnosis. Probabilistic inference provides CAPRI agents with the ability to diagnose failures with incomplete information. Previous systems for distributed fault diagnosis such as Netprofiler [67], Shrink [44], and Planetseer [98] cannot function without complete information from all or most of the parts of the system. Frequently when network failures occur, however, an agent cannot reach other diagnostic agents. In my experiments, I show that user agents can diagnose failures using probabilistic inference even when network failures prevent them from contacting regional agents.

Unlike previous work in distributed fault diagnosis, CAPRI agents can exchange probabilistic dependency knowledge in addition to observations and beliefs. The ability to communicate dependency knowledge allows agents both to take advantage of information about new diagnostic tests and to increase the accuracy and cost of diagnosis using existing tests. In my experiments, a knowledge agent learns more accurate dependency knowledge using Bayesian learning and communicates this knowledge to regional agents. Using this learned dependency knowledge, regional agents diagnose failures with both higher accuracy and lower cost.

In order to illustrate the effectiveness of probabilistic inference for modeling and diagnosing real-world failures, I measure the accuracy of diagnosis in my prototype implementation. Note that the purpose of this experiment is to demonstrate the ability for diagnostic agents to use distributed probabilistic inference for diagnosis and not to show that the probabilistic models I implemented are necessarily the best or more accurate.

In my experiments, a knowledge agent distributes probabilistic dependency knowledge to all other agents. Figure 9-1 illustrates the probabilistic dependencies that agents use for diagnosis in my prototype implementation. Initially, I manually specify the conditional probabilities for the dependency knowledge in the figure and provide the knowledge agent with this dependency knowledge. The knowledge agent advertises the capability to provide regional agents with dependency knowledge. As regional agents receive requests and require knowledge for diagnosis, regional agents request dependency knowledge from the knowledge agent. Similarly, when user agents start up, they request dependency knowledge from regional agents in turn. Note that user agents only receive a subset of all dependency knowledge from regional agents; user agents do not know about the dependencies for the information collected by specialist agents.

This section evaluates the accuracy and cost of probabilistic diagnosis in my experiments. In addition, this section discusses several advantages of a probabilistic approach, including the ability to compute the confidence of a diagnosis and the ability to compute the value of available diagnostic actions.

9.3.1 Improving accuracy and cost by learning

Cause of failure	Count	
DNS lookup	9395	(31%)
Web server	6106	(20%)
Local network	5103	(17%)
IP routing	2373	(8%)
Unknown	7023	(23%)

Table 9.2: Distribution of 30,000 failures

In order to evaluate the accuracy of diagnosis, I label 30,000 reported failures using a set of manually specified rules based on a posteriori information such as the number of consecutive failures to a web server and from a user. Figure 1 lists these rules. Table 9.2 illustrates the distribution of labels for the these failures. Though these labels may not always be correct, they provide a good starting point for estimating diagnostic accuracy. The strength of the CAPRI architecture is that if an expert develops more accurate rules for labeling failures, they can then train a Bayesian network using these rules to produce more accurate dependency knowledge for fault diagnosis. To demonstrate this feature of CAPRI, I trained the Bayesian network in figure 9-1 using the labels above on the first 20,000 failures. Appendix B.3 contains the conditional probabilities for dependency knowledge before and after learning. To evaluate the accuracy of diagnosis using this learned dependency model, I tested the learned dependency knowledge on the next 10,000 failures.

I evaluate the accuracy of diagnosis in terms of recall and precision. Recall refers to the number of correct diagnoses of a particular type divided by the total number of *actual failures* of that type, while precision represents the number of correct diagnoses of a particular type divided by the total number of *diagnoses* of that type. Accurate diagnosis requires both high recall and high precision. Low recall for a type of failure implies agents frequently misdiagnose failures of that type. Low precision for diagnoses of a given type implies that diagnoses of that type are frequently inaccurate.

Figure 9-3 shows the accuracy of diagnosis in terms of recall and precision for each candidate explanation before and after learning. Table 9.3 classifies failures by true cause and diagnosis using dependency knowledge before and after learning. Diagnosis using learned dependency knowledge increases overall accuracy from 86% to 97%. Notice that before learning, agents misdiagnosed many web server failures as DNS lookup or IP routing failures, resulting in low recall for web server failure diagnosis and low precision for DNS lookup and IP routing diagnosis. After learning, however, the recall of web server diagnosis and the precision of DNS lookup and IP routing diagnosis increased significantly, demonstrating the effectiveness of Bayesian learning for improving diagnostic recall and precision.

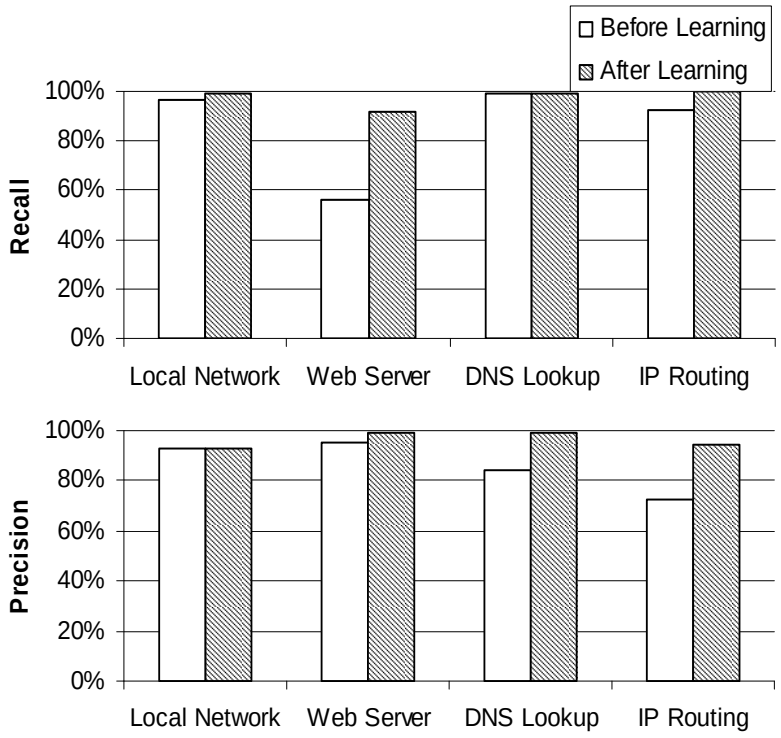

Figure 9-3: Learning dependency knowledge improves recall and precision

	Diagnoses before learning					
True cause	Local net	Web server	DNS lookup	IP routing	Total	**Recall**
Local net	1691	1	55	2	1749	**97%**
Web server	90	1163	527	279	2059	**56%**
DNS lookup	36	0	3068	0	3104	**99%**
IP routing	0	53	7	747	807	**93%**
Total	1817	1217	3657	1028	7719	
Precision	**93%**	**96%**	**84%**	**73%**		**86%**

	Diagnoses after learning					
True cause	Local net	Web server	DNS lookup	IP routing	Total	**Recall**
Local net	1738	11	0	0	1749	**99%**
Web server	97	1888	25	49	2059	**92%**
DNS lookup	36	1	3066	1	3104	**99%**
IP routing	0	3	0	804	807	**100%**
Total	1871	1903	3091	854	7719	
Precision	**93%**	**99%**	**99%**	**94%**		**97%**

Table 9.3: Diagnoses before and after learning

Also, notice that even though the probabilistic dependency model that agents use for diagnosis is relatively simple and leaves out many additional dependencies such as the user's operating system, link layer connectivity, and software dependencies, agents can still diagnose most failures with reasonable accuracy. This result demonstrates that probabilistic inference can frequently produce good results even without a complete model of dependencies. One reason for this robustness is that many common failures are relatively easy to diagnose. For example, an agent can diagnose failures having a Firefox "server not found" error code using an *Outbound Conn Test* with very high accuracy. If the `probeResult` of the *Outbound Conn Test* is `OK`, then it is a DNS lookup failure. Otherwise, it is a local network failure with high probability. If an agent has accurate tests and accurate probabilistic dependency knowledge for diagnosing common failures, the agent can diagnose such common failures with high accuracy even if the probabilistic dependency knowledge it has for other components is incomplete or inaccurate. Of course, if an expert has accurate dependency knowledge of these additional dependencies, the expert can introduce this new knowledge into the system to improve accuracy.

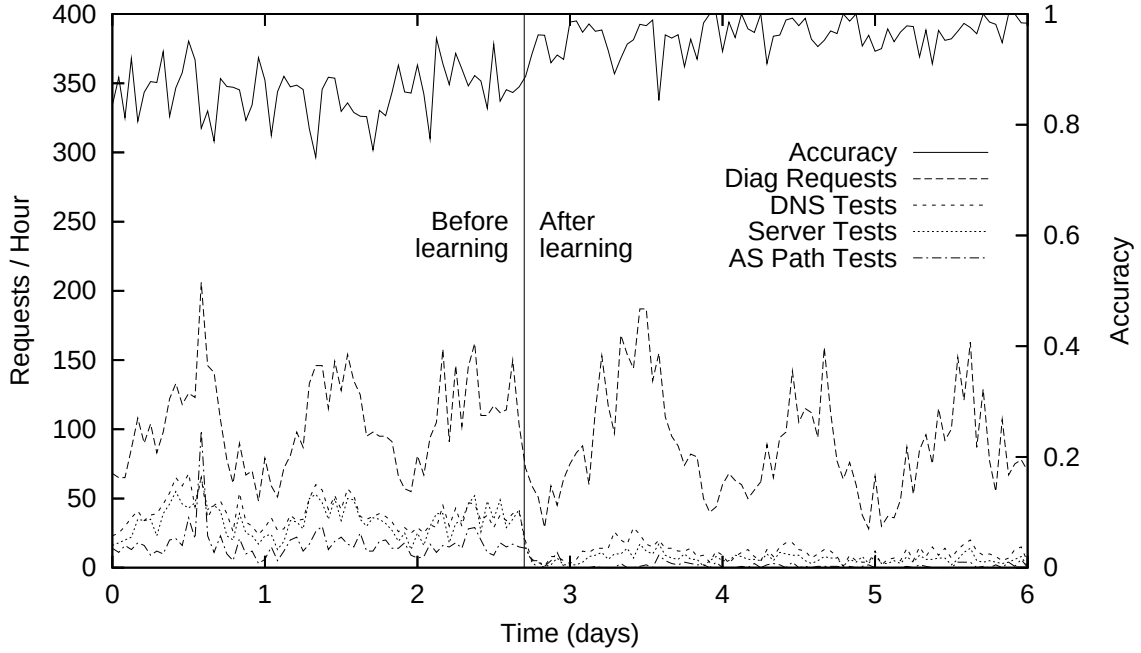

Figure 9-4: Learned knowledge improves accuracy and cost

Also, notice that learning new dependency knowledge not only improves accuracy, it reduces the cost of diagnosis as well. Using learned knowledge, the average accuracy of diagnosis increased from 79% to 94%, where accuracy is defined as the number of correct diagnoses divided by the total number of diagnoses. In addition, as regional agents retrieve the newly learned dependency knowledge from the knowledge agent, the average cost of diagnosis also decreases. Figure 9-4 shows average accuracy, the number of diagnosis requests to regional agents, and the number of specialist agent requests over time. After acquiring new dependency knowledge at approximately 2.7 days, the regional agents significantly reduce the number of requests they make to specialist agents while accuracy

improves. The number of diagnostic requests of all types decreases substantially after introducing the new dependency knowledge. This result suggests that the new dependency knowledge enables agents to diagnose failures more accurately with less evidence.

9.3.2 Predicting accuracy using confidence

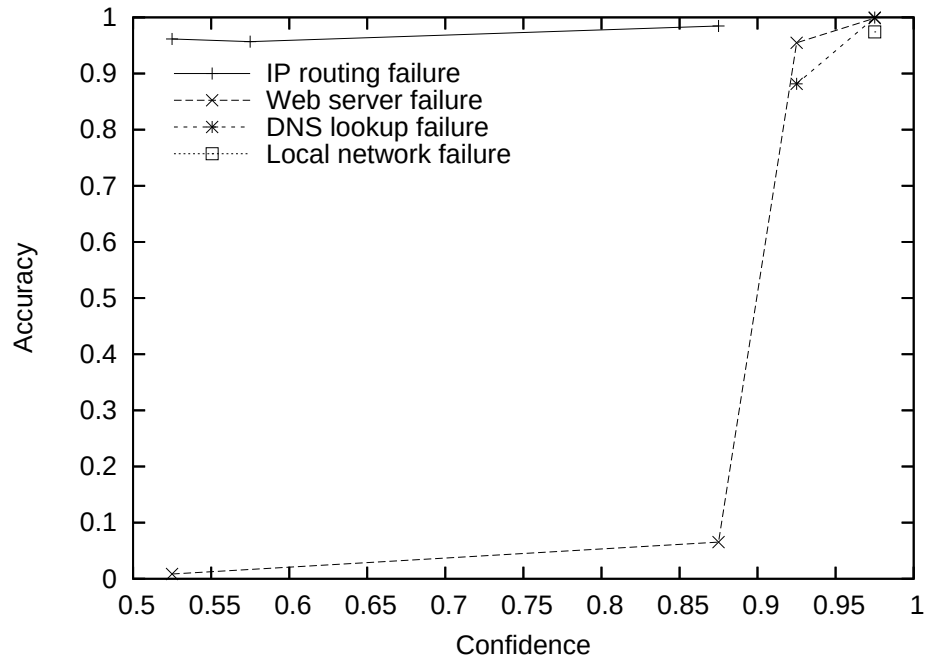

Figure 9-5: High confidence indicates high accuracy

Another benefit of probabilistic diagnosis is that it allows an agent to indicate its confidence in its diagnosis. A confidence metric helps a requester decide how much to believe the diagnosis and whether to request additional tests. Figure 9-5 plots the average accuracy of diagnosis for various confidence ranges for 55,866 diagnoses made between March 17 and April 10, 2007 using learned dependency knowledge. Each point on the plot represents the average accuracy of diagnoses with the specified explanation and confidence within the indicated range. For example, a diagnosis of DNS lookup failure with confidence between 0.95 and 1.0 is accurate nearly 100% of the time, while a diagnosis of DNS lookup failure with confidence between 0.90 and 0.95 is accurate 88% of the time. In this experiment, agents diagnose failures using a confidence threshold of 90%. Once an agent reaches 90% confidence, it does not perform any additional diagnostic tests. For clarity, the plot does not include points for confidence ranges with fewer than 50 diagnoses. For the data depicted in this figure, 86% of the diagnoses have confidence values over 90%. In general, greater confidence corresponds to greater accuracy, suggesting that a requester can use confidence as a prediction of diagnostic accuracy, especially for confidence levels greater than 90%.

This experiment also illustrates a deficiency in the probabilistic dependency model that agents use for diagnosis in this experiment. If agents had accurate dependency knowledge,

average accuracy would closely correspond to confidence. This is not the case in this experiment, however; confidence does not predict accuracy well for diagnoses with confidence levels less than 90%. This suggests that the dependency knowledge used by agents in this experiment does not accurately model all real-world dependencies. For example, most of the web server failure diagnoses with less than 90% confidence have a `ffoxErrorCode` of 14 (connection timed out), a `consecFailuresToServer` of 0, and a `dnsLookupResult` of `CORRECT`. According to the rules for labeling failures in Figure 1, this is an IP routing failure. Using learned dependency knowledge, however, regional agents misdiagnose these IP routing failures as web server failures. The fact that an agent has high confidence for incorrect diagnoses of certain types of failures suggests that the Bayesian network in Figure 9-1 does not completely capture all the dependencies in the system.

Unlike other systems for fault diagnosis that rely on static algorithms for diagnosis, however, CAPRI allows one to correct inaccuracies by introducing new dependency knowledge and new specialist agents. For example, one could create a new piece of dependency knowledge indicating that an *HTTP Connection* failure with `ffoxErrorCode` 14 and 0 `consecFailuresToServer` is most likely due to an *IP Routing* failure. One could define a new diagnostic test class with a property that represents `ffoxErrorCode` $= 14 \wedge$ `consecFailuresToServer` $= 0$ and then add `ffoxErrorCode`, `consecFailuresToServer`, and *IP Routing* `status` as parent properties. A new agent with this new knowledge may either directly offer its new knowledge to other agents, or it may simply offer a new service for inferring *IP Routing* `status` given an *HTTP Connection* failure for which `ffoxErrorCode` is 14 and `consecFailuresToServer` is 0. Existing regional agents could then automatically discover and take advantage of the new dependency knowledge or new service to correctly diagnose previously misdiagnosed failures.

Deciding whether to directly provide the new knowledge or to encapsulate it as a new service involves making a tradeoff between inference cost and communication cost. Directly providing the knowledge to regional agents can reduce the number of requests that regional agents need to perform for diagnosis, but increases the complexity of the failure dependency graphs used for inference. On the other hand, providing a new service may increase communication cost but does not increase the complexity of diagnostic inference at the regional agent.

9.3.3 Predicting change in accuracy using action values

Probabilistic dependency knowledge enables diagnostic agents in CAPRI to estimate the value of the information an action produces based on available evidence in a non–domain-specific way. The effectiveness of action selection based on action value depends on how closely action value corresponds to improvement in accuracy. In my experiments, agents compute action value as the expected change in confidence given the information produced by the action. To measure the degree of correlation between action value and improvement in accuracy, I compute diagnostic accuracy before and after each action that each regional agent performs. Figure 9-6 plots the average change in accuracy for actions having values within the indicated ranges for 2805 specialist requests from regional agents. On average, actions having value between 0 and 0.05 only improve accuracy by 2.1%, actions with value between 0.05 and 0.10 improve accuracy by 25%, and actions with value between

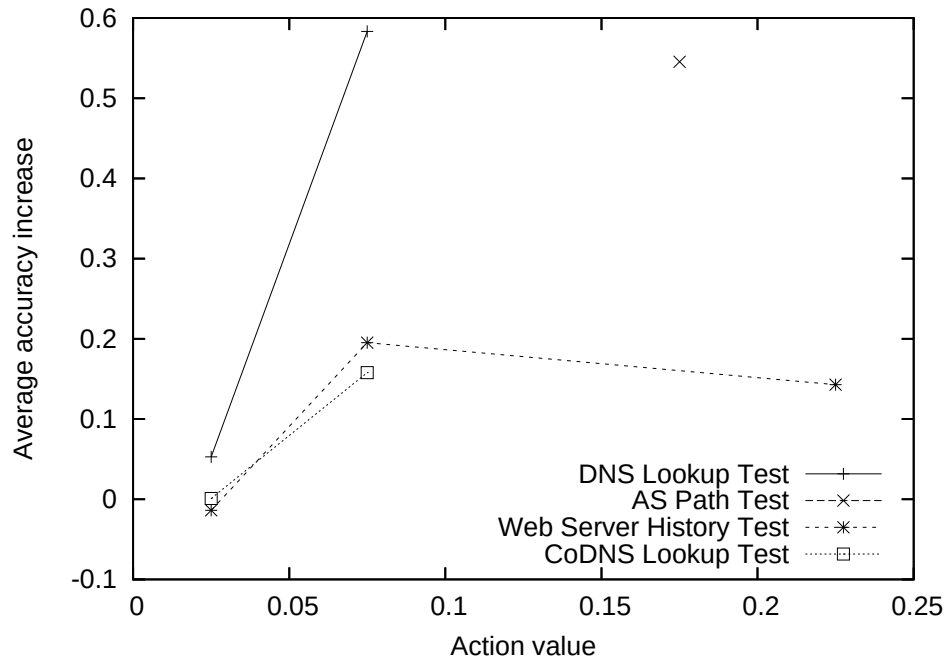

Figure 9-6: Action value predicts an action's effect on accuracy

0.15 and 0.20 improve accuracy by 55%. This result suggests that action value roughly predicts the expected change in accuracy for an action, with the notable exception of web server history tests having value between 0.20 and 0.25, which I examine in more detail below. In addition to helping agents decide what actions to take, action value might also help a requester decide whether to request additional information. For example, one might extend the CAPRI messaging protocol so that in addition to indicating the most probable cause of failure and the confidence of the diagnosis, a diagnostic response also includes a list of additional actions an agent can perform and their associated action values and costs.

Comparing action value and average change in accuracy can help identify inaccurate probabilistic dependency knowledge. For example, Figure 9-6 indicates web server history tests with value between 0.20 and 0.25 only improve accuracy by 14% on average. An examination of the log files reveals that in this experiment regional agents tend to overestimate the value of web server history tests for the diagnosis of HTTP connection timed out errors. Using learned dependency knowledge, a regional agent diagnosing an HTTP connection failure having `ffoxErrorCode` 14 (connection timed out), 0 `consecFailuresFromUser`, and *Outbound Conn Test* `probeResult` `OK` computes the value of a web server history test as 0.234 and requests this information from a web server history agent. In most cases, however, `consecFailuresToServer` equals 1, which actually decreases the regional agent's confidence in its diagnosis. This discrepancy between action value and average change in confidence indicates a deficiency in the regional agent's probabilistic dependency knowledge. In particular, the regional agent incorrectly assumes conditional independence between `consecFailuresToServer` and `ffoxErrorCode` in cases where `ffoxErrorCode` equals 14 and `consecFailuresToServer` equals 1. To remedy this deficiency, one might introduce a new diagnostic agent that has more accurate dependency knowledge relating

`ffoxErrorCode` and `consecFailuresToServer`, as described in Section 9.3.2.

9.4 Managing cost

Probabilistic inference allows agents in CAPRI to manage cost by diagnosing failures with incomplete information as well as using cached information to diagnose future similar failures. This section presents experimental results illustrating the ability of diagnostic agents to aggregate multiple similar failures to reduce the average cost of diagnosing multiple failures.

One of the key benefits of diagnosis in CAPRI is that a diagnostic agent *A* can communicate observations, beliefs, and probabilistic dependency knowledge to another agent *B* so that agent *B* can cache this information and diagnose future failures without having to contact agent *A* again. Note that by caching the intermediate observations used to produce a diagnosis and not just the final diagnostic result, agents can use this information to diagnose other similar but not identical failures in the future to efficiently aggregate multiple similar requests. This can both reduce the communication cost of diagnosis as well as provide additional robustness to network failures so that an agent can perform diagnosis even if it cannot contact any other agents in the network. In my experiments, I show that both regional and user agents can use the results of previously collected diagnostic tests to diagnose multiple similar failures using the same information without having to contact additional diagnostic agents.

9.4.1 Regional agents aggregate similar failures

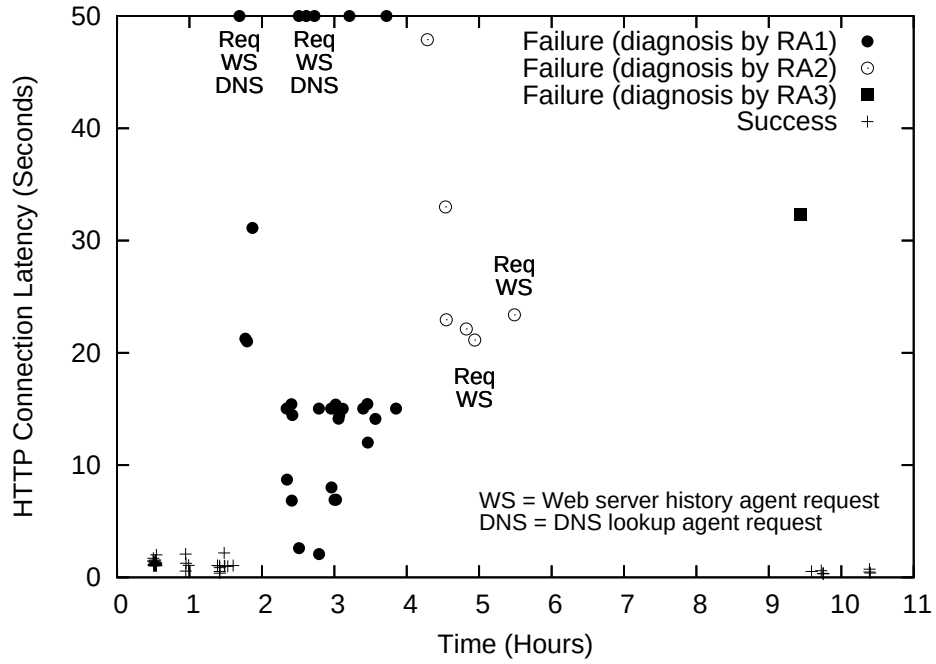

Figure 9-7: A regional agent aggregates multiple similar failures

Figure 9-7 illustrates a case of diagnosing multiple similar failures to the same destination hostname. Each point in the figure represents a connection attempt from a user to the destination web server hostname. In this case, multiple users experience failures connecting to the same destination hostname. Over the time period indicated in the figure, the IP address for the destination hostname changes. During the transition period, DNS lookup for the destination hostname fails for some time, and requests to the old IP address also fail for some time. This figure shows three regional agents diagnosing failures to connect to this destination from three different users. Initially, when RA1 first receives a diagnostic request at approximately 1.6 hours, it contacts two specialist agents to request two types of diagnostic tests, a web server history agent to request a belief about the `status` of the *HTTP Server* inferred from `consecFailuresToServer`; and a DNS lookup test agent to request a belief about the *DNS Lookup* `status` using the `dnsLookupResult` from a *Verify DNS Lookup Test*. Using this information, the agent diagnoses the failure as a DNS Lookup failure. It then caches this information to diagnose the next 10 failures. The *Verify DNS Lookup Test* information times out after one hour, and RA1 requests this information again. It then caches it for use in diagnosing the next several failures. Similarly, the second regional agent requests *HTTP Server* `status` beliefs from the web server history agent and uses this information to diagnose the next few failures. In this case, three diagnostic agents diagnose a total of 37 failures using only a total of six specialist agent requests, demonstrating the usefulness of caching and aggregation for reducing the cost of diagnosing multiple similar failures. Of the 37 failures, 26 are server not found errors, 6 are requests canceled by the user, 3 are connection reset errors, and 2 are connection refused errors. The regional agents diagnose all the failures as DNS lookup failures except the three failures between time 4.7 and 5.5 hours, which RA2 diagnoses as web server failures.

Note that the majority of failure requests in my experiments are for failures experienced by only one user. Therefore not every failure presents an agent with an opportunity for aggregation. As the number of users and failures increases, however, I expect that more opportunities for aggregation will arise.

This experiment also illustrates a limitation of diagnosis using my prototype implementation. In my prototype, agents do not distinguish among different types of DNS lookup failures. For example, a DNS lookup failure might result from one of a number of different causes, including a mistyped URL, a failure of the authoritative DNS server responsible for the web server's domain, a failure of the user's ISP's DNS server, a DNS server misconfiguration, or an IP routing failure between a user and a DNS server.

Unlike previous systems for fault diagnosis that can only distinguish among a limited set of explanations for a failure, however, CAPRI allows the introduction of new diagnostic agents with new capabilities. For example, an expert with the ability to distinguish among various types of DNS lookup failures might create a new diagnostic agent that offers a more detailed DNS diagnosis service. The expert might define additional network component classes such as *DNS Server*, additional diagnostic tests such as *Mistyped URL Test*, and new knowledge indicating the probabilistic dependencies among properties of these new classes and *DNS Lookup* `status`. The new DNS diagnosis agent may then advertise the ability to distinguish among various candidate explanations for DNS lookup failure. A user that wishes to obtain more detailed diagnosis of DNS lookup failures can then request additional diagnosis from the new agent.

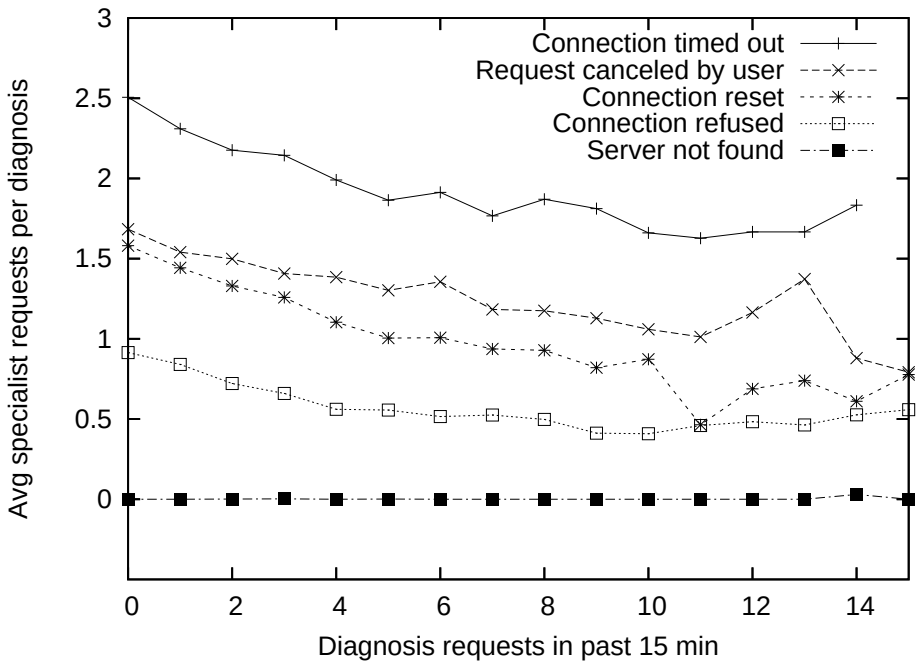

Figure 9-8: Aggregation reduces the cost of diagnosis as the rate of failures increases.

To quantify the effect of aggregation, I plot the number of specialist requests that regional agents make as a function of the number of recent diagnostic requests they have received. An important reason to aggregate similar requests is to reduce the cost of diagnosing high-impact failures that affect a large number of users. Figure 9-8 shows that as the rate of requests a regional agent receives increases, the average number of specialist agent requests that agent makes decreases. The figure plots the average number of specialist requests for 30,000 regional agent diagnoses from March 5 to March 16, 2007 as a function of Firefox error code and the number of requests an agent has received in the past 15 minutes. For clarity, the figure only plots points corresponding to averages of 15 or more diagnoses. An agent that has received many requests in the past 15 minutes is more likely to have cached information useful for diagnosing a failure. This result suggests that caching recent failures is an effective way to reduce the cost of diagnosing multiple similar failures, especially when many such failures occur in a short period of time.

9.4.2 User agents aggregate similar failures

User agents also aggregate similar failures. In particular, if a user experiences multiple simultaneous failures to different destinations, there is a high probability that the failures are all due to a local network connectivity problem, such as a disconnected network cable. In such cases, a user agent diagnoses all of these failures without requesting additional diagnosis from regional agents. It is important for user agents to diagnose such failures efficiently because a user disconnected from the network cannot contact a regional agent for additional diagnosis. Figure 9-9 illustrates such a case of aggregation at a user agent. In this case, a user experiences eight consecutive failures to eight different destination web

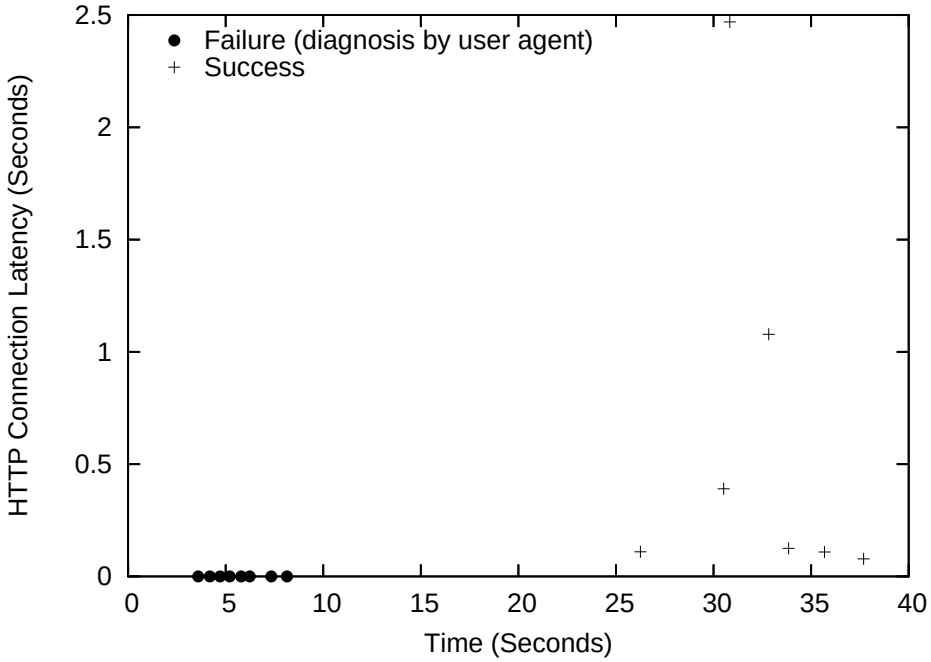

Figure 9-9: A user agent aggregates multiple similar failures

servers within a space of less than ten seconds. The user might be a mobile user who just turned on their computer and does not have network connectivity for several seconds until they receive an IP address using DHCP. When these failures occur, the user agent conducts local tests, including outbound connectivity tests and computing the number of consecutive failures from the user. Using the results of these tests, the user agent diagnoses all of these failures as local network failures. This example illustrates how aggregation of similar failures at the user agent can greatly reduce the number of diagnostic requests that regional agents must handle. In addition, this case demonstrates the ability to diagnose failures with incomplete information using probabilistic diagnosis, which is essential when network failures prevent agents from collecting additional diagnostic information.

9.5 Extensibility

The experiments I conduct illustrate extensibility in several ways. Firstly, they show the ability for agents to add new dependency knowledge. Once the knowledge agent learns new dependency knowledge, it can provide other agents with these new updated probabilities. As demonstrated in the previous section, this both improves the accuracy of diagnosis and also reduces cost.

Another type of extensibility is the ability to support new diagnostic tests in order to provide new functionality, improve accuracy, or reduce costs. My prototype implementation makes it easy to wrap existing diagnostic tests to produce new diagnostic agents. In my experiments, I introduced a second specialist agent for diagnosing *DNS Lookup* components that uses the CoDNS service [69]. CoDNS is an existing service on Planetlab that caches DNS lookups to provide faster response times. In order to add a new type of

specialist agent that can perform CoDNS lookup tests, I performed the following steps:

1. I extended the component ontology by defining a new *CoDNS Lookup Test* class and a property `codnsLookupResult`. I defined these in a new component class and property definition file. See Appendix B for the actual definitions. Other diagnostic agents automatically request these definitions using the URI

   http://capri.csail.mit.edu/2006/capri/planetlab

   when they encounter the new component class and property.

2. I deployed a CoDNS specialist agent on Planetlab that advertises the capability to produce beliefs about *DNS Lookup* `status` from the `codnsLookupResult` of a *CoDNS Lookup Test*. Appendix B includes the actual service descriptions. Since CoDNS caches DNS lookups among Planetlab nodes, an agent can perform a *CoDNS Lookup Test* faster and using less network resources than a regular *Verify DNS Lookup Test*. Therefore CoDNS specialist agents can advertise a lower cost for performing CoDNS lookup tests.

3. I added new dependency knowledge to the CoDNS specialist agent. The CoDNS agent then provides the new knowledge to other agents so that they can use cached *CoDNS Lookup Test* information and compute the value of information provided by a `codnsLookupResult`. Appendix B contains this dependency knowledge.

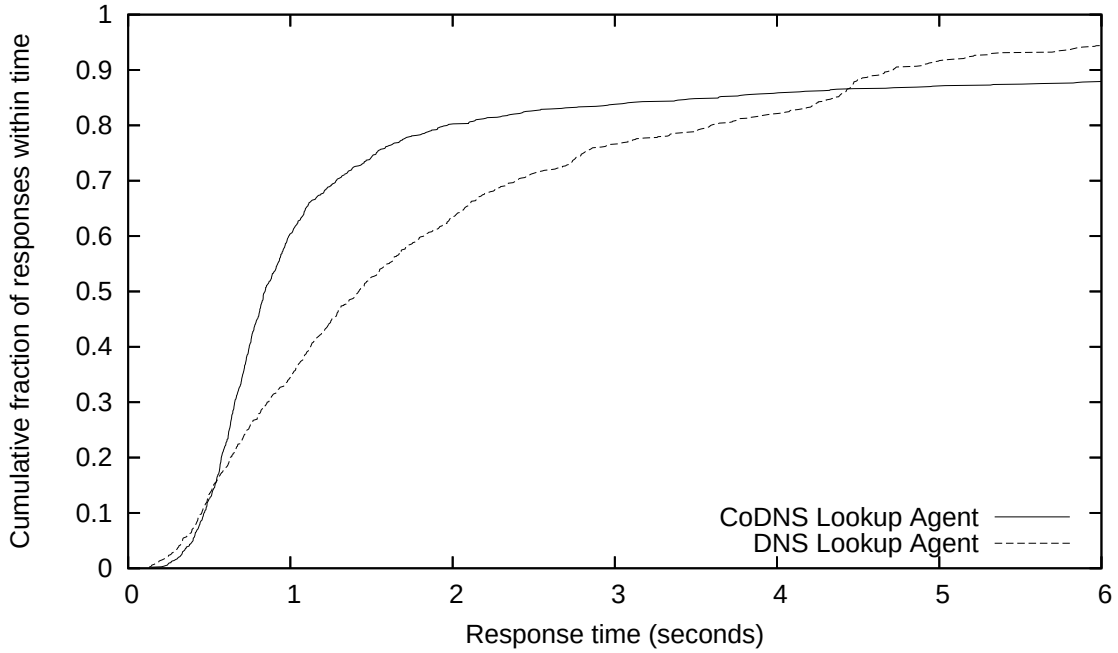

Figure 9-10: CoDNS lookup agents respond to requests more quickly than DNS lookup agents

After introducing CoDNS agents, regional agents discover the new specialist agents through the agent directory. Because the CoDNS agents advertise their services at lower

cost than regular DNS lookup agents, regional agents prefer them over regular DNS lookup agents. Figure 9-10 depicts the cumulative fraction of responses as a function of time. The median time for a CoDNS agent to perform a test is 0.84 seconds, compared to 1.21 seconds for regular DNS lookup agents, an improvement of 41%. Notice that the lines in the figure cross at about 0.5 and 4.5 seconds. This reflects the different distribution of response times for CoDNS and DNS lookup tests. The CoDNS agent requires more time to produce a response than the DNS lookup agent for the fastest 15% and the slowest 15% of responses. The DNS lookup agent might perform slightly better than the CoDNS agent for the fastest 15% of responses because it does not incur the overhead of using the CoDNS service. For the slowest 15% of responses, the CoDNS agent requires significantly more time to produce a response. This slowness might be caused by heavy load on the CoDNS agent generated by other processes on the Planetlab node.

The different cost distribution of different diagnostic tests suggests that it may be valuable for agents to represent diagnostic cost as a distribution rather than simply as a scalar value. For example, an agent wishing tightly bound the time required to perform an action may select actions on the basis of 95th percentile cost, while an agent with no time constraints might select actions based simply on expected cost. In future work, one might extend the CAPRI service advertisement language to support such cost distributions.

This case also demonstrates the ability of agents to automatically compute the value of information provided by a service and to select additional diagnostic actions when necessary. A *CoDNS Lookup Test* does not provide as much information as a regular *Verify DNS Lookup Test*. In particular, a *CoDNS Lookup Test* cannot determine whether a hostname is an alias or not, which is a strong indicator that the web site uses Akamai and that different users will get different IP addresses. Sometimes a *CoDNS Lookup Test* is sufficient to diagnose a failure, but in most cases an agent determines that the *CoDNS Lookup Test* does not provide enough information and requests a regular *Verify DNS Lookup Test* as well.

9.6 Summary of results

The prototype implementation described in this thesis and the experiments I conduct using this prototype illustrate several important benefits of the CAPRI architecture, including the ability to distribute diagnostic information among multiple agents, the ability to perform diagnosis with incomplete information, extensibility to support new dependency knowledge and new agents, and the ability to reduce the cost of diagnosing multiple failures using caching and aggregation.

Additionally, this experiment demonstrates the ability for diagnostic agents to aggregate observations collected from multiple users to produce new observations and dependency knowledge. CAPRI enables distributed processing of information so that one agent can aggregate and compose information to produce new information that other agents can use for diagnosis. In my experiments, web server history agents receive notifications of approximately 500,000 *HTTP Connection* observations per day from over 10,000 users to compute `consecFailuresToServer` and produce beliefs about *HTTP Server* `status`. A knowledge agent collects observations of *HTTP Connection* failures and `consecFailuresToServer` observations to learn new probabilistic dependency knowledge.

Chapter 10

Conclusion and Future Work

This thesis proposes a new way of looking at the problem of Internet fault diagnosis. Rather than attempting to develop specialized, isolated systems for the diagnosis of different types of failures, I seek to create a common framework for fault diagnosis in the Internet to enable cooperation among multiple diagnostic systems. The generality and extensibility of CAPRI allows the accumulation of new diagnostic capabilities and dependency knowledge to improve the quality of diagnosis over time.

This thesis addresses several challenges of developing such a common architecture. Firstly, CAPRI addresses the challenge of representing diagnostic information by defining a language for defining new component classes and properties in a general and extensible way. This representation enables CAPRI to support the diagnosis of new types of network components and new classes of diagnostic tests. Secondly, CAPRI addresses the challenge of discovering the capabilities of new diagnostic agents by providing a service description language for agents to specify the inputs and outputs of their diagnostic services. Using this service description language, agents can automatically take advantage of new diagnostic capabilities offered by other agents. Thirdly, CAPRI addresses the challenge of distributed inference by providing diagnostic agents with a general, non–domain-specific procedure for dynamically constructing failure dependency graphs from diagnostic information to perform fault diagnosis. This procedure allows agents with different capabilities and different diagnostic information to cooperate to perform distributed diagnostic inference while managing cost. I demonstrate the capabilities of the CAPRI architecture with a prototype implementation for the distributed diagnosis of real-world HTTP connection failures using several types of diagnostic agents.

Several important features of CAPRI include:

1. The separation of representation of diagnostic information in a component graph from the probabilistic dependency knowledge used to make inferences based on the information (see Chapter 4). This representation enables agents to apply new dependency knowledge for inference and to distribute the responsibility for collecting evidence, learning dependency knowledge, and performing diagnostic inference.

2. Dynamically specifying the set of candidate explanations in a diagnostic request based on the candidate explanations specified in a diagnosis service advertisement

(see Section 3.2). This allows the introduction of new diagnostic agents that can distinguish among new possible causes of failure, and it gives requesters the ability to specify the candidate explanations that they care about.

3. Dynamically selecting actions to perform in a general, non–domain-specific way based on the inputs and outputs specified in a service description (see Section 7.8). This procedure makes it possible for agents to discover new diagnostic services and use them for diagnosis.

4. The ability to use probabilistic inference to diagnose failures with incomplete information. The procedure described in Section 7.8 allows an agent to provide a diagnosis with incomplete information even when it cannot contact other agents or conduct certain tests.

10.1 Limitations and future work

Many open questions and areas of future work remain. In this thesis I make many design choices based on simplicity and ease of deployment. In some cases, however, alternative design choices might result in improved expressiveness or accuracy. Below I discuss the strengths and limitations of CAPRI, possible improvements, and future research directions.

10.1.1 Scalability

A major strength of the CAPRI architecture is that it supports the addition of new agents, new services and new dependency knowledge. In order for agents to effectively take advantage of new services and knowledge, however, the architecture must scale as the number of agents and knowledge in the system increases. This thesis considers three types of scalability: scalability to support many diagnostic requests, scalability to support a large number of available diagnostic services, and scalability to support a large amount of dependency knowledge. Below I discuss these scalability issues and describe possible areas of future work to better understand the scalability of CAPRI.

The first type of scalability is the ability to support a large number of diagnostic requests. In my experiments I show that agents in CAPRI can aggregate multiple similar failures using cached information to reduce the cost of diagnosing many similar failures. Secondly, I show how specialist agents use input restrictions to limit the requests they receive and distribute requests across multiple agents. Another technique that may reduce the cost of diagnosing multiple similar failures is evidence propagation. An agent that has evidence useful for diagnosing a failure may propagate that evidence to other agents that it believes may benefit from that evidence. Evidence propagation might be implemented as a type of notification subscription, or the evidence might simply be piggybacked on a response message. Such evidence propagation may increase the short-term communication costs of diagnosis by generating additional messages, but evidence propagation can potentially reduce the load of diagnosis on an agent with information that many other agents want and increase the speed with which failures are diagnosed in the long term. An area of

future work is to examine this tradeoff and determine when it is appropriate to propagate evidence to other agents.

Second is scalability in terms of available services. As the number of services that agents can choose from increases, it becomes more costly to compute the value of all services and choose among them. Agents in CAPRI address this issue by aggregating multiple specialist services together into a more general service. CAPRI service descriptions enable agents to achieve such aggregation of services using both input restrictions and requester restrictions. For example, in my prototype implementation, regional agents hide the complexity of specialist agents from user agents by aggregating all the specialist services into a single HTTP connection diagnosis service. Aggregation of services reduces the number of other agents and services that each agent needs to know for diagnosis, but an agent that knows about fewer services might not be able to achieve the same level of accuracy as an agent with access to more services. In addition, having multiple intermediate generalist agents may increase the communication costs of distributed diagnosis because more agents might need to be contacted. On the other hand, a generalist agent can also reduce cost if it can cache multiple similar requests. Effectively constructing a scalable topology for distributed diagnosis requires consideration of all of these factors. An area of future work is to conduct experiments to better understand these tradeoffs for real-world diagnosis.

A third issue is scalability in terms of dependency knowledge and failure dependency graphs. Additional dependency knowledge and more complex failure dependency graphs can improve diagnostic accuracy, but at the cost of additional computation to perform inference. Agents in CAPRI manage such costs by decomposing dependencies into conditionally independent parts and exchanging information using belief propagation. In reality, few component properties are truly conditionally independent, though in practice we can safely assume independence in many cases where the dependencies are sufficiently weak. To take an extreme example, two ISPs in different cities may fail simultaneously due to a sudden hurricane that hits both cities, but since the probability of such an occurrence is extremely low, it is probably safe to model the status of the two ISPs as being conditionally independent. An area of future work is to quantify the tradeoff between accuracy and inference cost to determine when it is safe to assume independence.

10.1.2 Extending the prototype implementation

The prototype implementation in Chapter 8 demonstrates the ability to diagnose HTTP connection failures using the CAPRI architecture. In order to demonstrate the generality and extensibility of CAPRI, it would be valuable to implement additional agents with new capabilities for diagnosing new types of failures.

For example, we might consider more detailed diagnosis of DNS lookup failures. A DNS lookup failure may be caused by an incorrectly typed URL, a malfunctioning or misconfigured DNS server, or a failure to reach the DNS server. One might develop additional agents to distinguish among these cases, or use the information provided by existing agents. For example, an agent might attempt to infer whether a hostname is spelled correctly by requesting an observation from the stats agent to determine how many previous users have attempted to connect to that hostname. Alternatively, one might perform multiple verify DNS lookup tests using multiple agents in different parts of the network and determine

whether the results are consistent. Or if one has access to the configuration of the DNS server, an agent might analyze the configuration file to identify any misconfigurations.

Another type of failure one might diagnose is IP routing failures. If one has the ability to perform traceroutes from different locations in the Internet, one can develop diagnostic agents that can infer the status of an IP route from the result of multiple traceroutes. This approach is similar to that of Planetseer [98]. In addition, we can consider the diagnosis of other applications, such as FTP connection failures and SMTP failures.

Another direction of future work is to consider alternative technologies for implementing diagnostic agents in order to achieve better performance or to support additional platforms. The primary objective of my experiments in this thesis is to demonstrate the benefits of the CAPRI architecture, and not to optimize the performance of diagnostic agents. We can improve the performance of diagnostic agents in several ways. In this thesis I implemented CAPRI diagnostic agents using two technologies: Firefox extensions and Apache mod_python handlers. A major cost of diagnosis in CAPRI is the computation of service value. Computing the value of a service requires multiple inferences about the expected value of information that the server produces. The Python OpenBayes module for Bayesian inference that agents currently use is quite slow, however. For improved performance, one might implement diagnostic agents natively in C or C++, or replace the OpenBayes module with a module written in C. For example, the SMILE Bayesian inference library can provide roughly a factor of 10 to 100 times speed improvement over the Python OpenBayes inference module.

Though in my experiments I found HTTP and XML suitable for the implementation of diagnostic agents, in future work we may also consider implementations of CAPRI agents using other technologies. Custom protocols might yield a performance advantage or provide other benefits.

10.1.3 Representation of diagnostic information

CAPRI provides diagnostic agents with a probabilistic representation for diagnostic information. This representation is general and extensible to support a wide range of observations, beliefs, and dependency knowledge about network components and diagnostic tests, but this representation has some limitations.

As mentioned in Chapter 4, one limitation is that agents cannot represent probabilistic beliefs about relationship properties. This limitation arises from the Bayesian model of network dependencies. Bayesian inference does not support probabilistic beliefs about the structure of a Bayesian network. In many cases an agent can work around this limitation using probabilistic beliefs about the properties of a component, however.

Also, in CAPRI the range of properties is limited to individual values and individual components, and not sets of values or sets of components. In some cases one may wish to describe a property with multiple values. For example, an agent might wish to describe a router that has multiple neighboring routers using a relationship property whose value is the set of neighboring router components. One way to overcome this limitation is to enhance component class and property definitions to allow the definition of properties with multiple values. In addition, agents will also need the ability to reason about properties with multiple values and to identify components within a set of multiple components.

Agents in CAPRI make a tradeoff by communicating compiled knowledge rather than deep knowledge about the network. The advantage of communicating compiled knowledge is that other agents do not need domain-specific information to diagnose failures. On the other hand, using compiled knowledge may throw away information useful for diagnosis. To address this problem, CAPRI agents might communicate detailed observations about properties for which they do not have dependency knowledge. If new agents with new dependency knowledge for these observations join the system in the future, other agents may then use the observations for diagnostic reasoning.

CAPRI agents represent beliefs and probabilistic dependencies using conditional probability tables. Though conditional probability tables can express any probabilistic function over discrete variables, they have two limitations. Firstly is that they require discrete variables. In practice this is often not a major problem because one can usually discretize continuous variables without losing much accuracy. Another drawback is that they can be extremely verbose, especially for conditional probability tables with many parent variables or variables with a large number of possible values. To address these challenges, one might consider representing probabilities using Gaussian or other probability distributions [65].

10.1.4 Probabilistic inference

This thesis demonstrates the advantages of probabilistic inference as a general technique for distributed fault diagnosis. Some of the advantages of probabilistic diagnosis include the ability to diagnose failures with incomplete information, the ability to diagnose failures without domain-specific knowledge, the ability to take into account new probabilistic dependency knowledge, the ability to dynamically compute the value of actions, and the ability to decompose dependencies and distributed dependency knowledge across multiple agents. A valuable area of future work is to examine the limitations of probabilistic inference in CAPRI and investigate possible improvements.

One area of future work is to evaluate probabilistic inference using more complex dependency models than the ones in my prototype implementation. The prototype implementation described in this thesis diagnoses failures using a polytree-structured Bayesian network with no undirected cycles. This Bayesian network omits some dependencies however, such as software dependencies and link layer dependencies. An area of future work is to examine the impact of adding such additional dependencies. Standard belief propagation is not designed for inference when undirected cycles are present in a Bayesian network; an interesting question is whether loopy belief propagation [17] is effective for real-world network fault diagnosis.

Another possible research direction is to examine the effectiveness of dynamic Bayesian networks (DBN) for diagnostic inference in CAPRI. Section 7.5.1 describes how to apply DBNs for inference in CAPRI. Inference using DBNs allows agents to take into account temporal models of failure, though with significantly greater computational cost. An area of future work is to evaluate the tradeoffs in accuracy and cost that DBNs can provide for real-world fault diagnosis.

In my experiments the knowledge agent periodically learns new dependency knowledge offline. One possible area of future research is to investigate the advantages and drawbacks of other approaches for learning new dependency knowledge, such as on-line learning and

reinforcement learning. Because probabilistic dependencies might change over time in the Internet, another question is how frequently to learn new dependency knowledge. Agents must make a tradeoff between learning more frequently based on less data, or learning less frequently using more data.

Another limitation of CAPRI is that by its very nature, probabilistic diagnosis according to a general and dynamic architecture for fault diagnosis requires additional overhead and computational cost compared to a specialized diagnostic system. Exact Bayesian inference can be computationally costly, especially in DBNs as the number of nodes in a Bayesian network increases. If computers continue to become faster and less expensive, however, then the computational cost of diagnosis will decrease over time. In addition, one might reduce the computational cost of diagnosis using algorithms for approximate inference [65, 26].

10.1.5 Service description and discovery

CAPRI agents look up and retrieve service descriptions using a centralized service directory. A centralized directory is vulnerable to failure and attack, however, so one might consider building a more decentralized and distributed directory service. Implementing such a distributed directory presents many challenges, however, including maintaining consistency and the distribution of services among multiple directory servers. One might also examine enhancements to the service advertisement and lookup protocol, such as a more sophisticated service indexing system and a lookup protocol that enables agents to retrieve only services that have changed to reduce the size of service directory messages. Another area of research is to consider other modes of service discovery as well, such as using link layer broadcast messages.

In future research, one might consider the advantages of a richer service description language. As a first step, in this thesis I define services using a flat namespace. Services do not specify their relationship to other services. An interesting area of future work is to examine the potential benefits of organizing services into classes and subclasses and explicitly describing the relationships among different services to create a service ontology. Such a service ontology might enable agents to better reason about how to compose and aggregate multiple services.

10.1.6 Action selection

The prototype CAPRI agents described in this thesis select actions myopically according to their expected value of information. An area of future work is to examine more sophisticated action selection algorithms that can take into account additional factors such as the value of performing multiple simultaneous actions, the expected future distribution of failures, the expected future cost of diagnosis, and the trustworthiness of the agent providing the information.

Another question is whether an action selection algorithm that performs multiple actions simultaneously in parallel can improve the cost and accuracy of diagnosis. Currently agents only select single actions myopically; implementing an action selection algorithm

that takes into account the expected benefits and costs of multiple actions may produce better results.

Another challenge is to take into account issues of security, trust, and authentication when deciding what services to use and which agents to contact. As the number of specialist agents increases, agents need to be able to decide which ones are trustworthy and which ones are malicious. In addition, some agents may provide more accurate information than others. Especially when different agents have conflicting information, agents need to be able to decide which other agents to contact. Possible approaches to this problem are to learn the value of the services other agents offer, to construct a web of trust, or to develop a reputation system whereby accurate and reliable agents gradually accumulate a reputation.

10.1.7 Cost and economics

This thesis assumes that the costs of diagnostic services are set exogenously. Costs may reflect the expected time to perform a diagnostic test, the network resources used, or may be tied to actual monetary payments. Quantifying the cost of diagnostic tests can be difficult, however, especially since the cost of an action may not be known ahead of time. One area of future work is to examine how agents might set these costs. The cost of performing a test may be borne by not only the agent performing the test, but other entities in the network as well. In order to avoid concentration of costs, it is important to consider all of those who may be affected by a test.

Another possible area of research is to investigate other cost models. CAPRI limits cost descriptions to scalar integer values, but in some cases it may be more accurate to specify a cost distribution instead. For example, an agent wishing to tightly bound the time required to perform an action may select actions on the basis of 95th percentile cost, while an agent with no time constraints might select actions based simply on expected cost.

Another area of future work is to study the incentives and economics of a marketplace for diagnosis. In my prototype implementation, the cost of services is fixed. As an alternative, agents might dynamically set the cost of a service. An agent under heavy load may choose to increase the cost of its service temporarily to reduce the number of requesters, for example. Such dynamic pricing may also lead to competition among multiple diagnostic agents for servicing requests and create a marketplace for diagnosis.

A related question is to evaluate the costs and benefits of diagnosis in terms of the repair actions that requesters take on the basis of diagnostic results. The accuracy and value of a diagnosis should ultimately be calculated in terms of the effects of the repair actions that the requester takes. Certain types of misdiagnoses may have more serious consequences than others. For example, if a network administrator brings down their entire network in order to repair a failure based on incorrect information in a diagnostic response, then a misdiagnosis can be very costly. On the other hand, if the action taken by the requester is the same as the one that they would have taken given the correct diagnosis, then the misdiagnosis does not cause any harm. An area of future work is to consider how to incorporate this type of reasoning into diagnostic agents.

10.1.8 User interface and user feedback

Another research direction is to consider how to provide better feedback to users and to support more interactive diagnosis. The prototype implementation in this thesis does not give users the ability to provide feedback to diagnostic agents or adjust the parameters of diagnosis. To increase interactivity, one might modify the user interface to enable users to request additional tests and adjust the confidence threshold, budget, and expiration time of diagnosis according to their needs. In addition, one might modify the user interface to allow users to provide feedback about the results of diagnosis. Such feedback might allow agents to improve their accuracy or provide a mechanism for distinguishing among trustworthy and untrustworthy agents.

Another area of research is to examine the benefits of providing diagnosis requesters with more information about the tradeoffs of specifying different parameters in a diagnostic request. For example, a diagnosis requester might provide more or less information in the request, or specify different confidence thresholds, budgets, and expiration times. A recipient of a diagnostic response does not know whether the diagnosis provider had the ability to produce a more accurate diagnosis but chose not to because it had sufficient confidence, expended its budget, or exceeded the expiration time. To help a requester better determine the degree to which they should believe the diagnosis and the tradeoffs of requesting additional diagnostic tests, perhaps an agent could estimate the potential additional confidence and cost of diagnosis with an increased confidence threshold, budget, or expiration time. A diagnostic agent might also specify in a response how much additional confidence it can provide in a diagnosis given various combinations of inputs. An agent might estimate its expected confidence with additional inputs by computing the expected value of information of the additional inputs. A diagnosis provider might also provide more information about what actions it took and what actions were available to give the requester more feedback about the process of diagnosis.

Another research direction is to consider giving requesters more control over the actions taken in a diagnosis. For example, a diagnosis provider might provide a requester with the set of possible actions it has available for diagnosis and the value and cost of each action. The requester might then explicitly specify which tests it wishes to conduct and which ones it does not. Allowing requesters to explicitly specify tests to perform introduces additional challenges, however. A requester may not have all the information necessary to decide what tests to perform, and exposing the set of available actions to requesters makes it more difficult to introduce new specialist agents and to change the way in which a diagnosis provider operates.

10.2 Conclusion

In this thesis, I present a new approach to fault diagnosis based on a common architecture for communicating diagnostic information and performing diagnostic reasoning. By providing modular interfaces for diagnostic agents, CAPRI enables cooperation among multiple specialized diagnostic agents. I hope that this general architecture encourages future researchers in Internet fault diagnosis to rethink the way they develop diagnostic

tools. I hope that rather than developing monolithic tools for diagnosis using domain-specific knowledge, future researchers will consider more modular tools and share their diagnostic knowledge to encourage reuse and cooperation. As researchers develop new and more sophisticated diagnostic agents, the accuracy and power of diagnosis will improve. Combining new diagnostic agents and new knowledge in novel ways may produce new and unexpected capabilities. If a common architecture for fault diagnosis such as CAPRI becomes widely adopted, both users and network administrators will benefit from faster, cheaper, and more accurate diagnosis of network failures. This thesis is just the first step towards this goal, however, and much work remains for future researchers to address.

Appendix A

CAPRI Language Specifications

CAPRI provides diagnostic agents with an ontology language for defining component and diagnostic test classes and their properties, a language for representing diagnostic information and diagnostic messages, and a language for representing service descriptions. This appendix contains the specifications for each of these languages expressed using XML.

A.1 Component ontology language

CAPRI provides agents with a component ontology language for defining component and diagnostic test classes and their properties. This section describes the XML serialization of this ontology language.

A.1.1 Component class and property definition files

Component and diagnostic test classes are defined in component class and property definition files. A file may contain multiple class and property definitions. Every component class and property definition also has an associated unique URI that refers to the class or property's corresponding component class and property definition file.

A component class and property definition file contains a single root element, `componentOntology`. The `componentOntology` element may contain any number of `componentClass`, `descriptiveProperty`, and `relationshipProperty` child elements. The order of the component class and property definitions within the file does not matter.

A.1.2 Defining component classes

A component or diagnostic test class definition is specified using the `componentClass` element. The `name` attribute of `componentClass` element specifies the component class name as a URI. The `componentClass` element may have zero or more `subclassOf` and `idProperty` child elements listing its superclasses and identifying properties, respectively. The order of the `idProperty` elements does not matter. Each `subclassOf` property has a `name` attribute whose value is the URI of a superclass for the component or diagnostic test class. Each `idProperty` child element has a `name` attribute whose value is the URI

of an identifying property for the component or diagnostic test class. For example, the component class definition for `HTTP Connection` is as follows.

```
<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
  <idProperty name="http://capri.csail.mit.edu/2006/capri/common#destHash" />
  <idProperty name="http://capri.csail.mit.edu/2006/capri/common#connTime" />
</componentClass>
```

A.1.3 Defining properties

CAPRI supports the definition of both descriptive properties and relationship properties.

Defining descriptive properties

A descriptive property is defined using a `descriptiveProperty` element whose `name` attribute specifies its full name as a URI. A descriptive property may also optionally contain a `range` child element specifying the range of possible values that this property can have. The `range` element contains a list of two or more `entry` elements. The `value` attribute of an `entry` element specifies a possible value that the property can have. The order of the entries does not matter. Note that to facilitate extensibility to support new properties of existing components, the set of allowable properties of a component is not restricted. For similar reasons, the domain of a property is also unrestricted so that any component may have any defined property. For example, the property definition for `status` is given below.

```
<descriptiveProperty name="http://capri.csail.mit.edu/2006/capri/core#status">
  <range>
    <entry value="OK" />
    <entry value="FAIL" />
  </range>
</descriptiveProperty>
```

Defining relationship properties

A relationship property is defined using the `relationshipProperty` element, whose `name` attribute specifies the full name of the property as a URI. A `relationshipProperty` element must also have a `range` attribute whose value is the component class name of the class of component to which this property refers. A `relationshipProperty` element may optionally specify a `symmetricProp` attribute whose value is the name of a symmetric relationship property. To understand the meaning of a symmetric property, consider a relationship property `prop` has with range A and symmetric property `symProp`. This definition implies that for every component X with a relationship property `prop` referring to a component Y, an agent can infer that component Y has a relationship property `symProp` that refers to X. As an example, the definition of the `asHopTest` relationship property is given below.

```
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asHopTest"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#asHop"
/>
```

A.2 Representation of diagnostic messages

This section describes the representation of diagnostic information and diagnostic messages using XML.

A.2.1 Representing diagnostic information

Agents can communicate four types of diagnostic information: observations, beliefs, likelihoods, and dependency knowledge.

Representing observations

Agents express observations using an `observation` element. The observation has five attributes providing metadata about the observation: an `id` attribute, a `time` attribute indicating the time the observation was made in milliseconds since Jan 1, 1970 UTC; an `originator` attribute indicating the URI of the agent that originally made the observation; a Boolean `cached` attribute indicating whether the observation was cached or not; and an `expires` attribute indicating the time the information expires in milliseconds since Jan 1, 1970 UTC.

An `observation` element has exactly one child element, either a `component` element or a `test` element, depending on whether the input component is a network component or a diagnostic test. The `component` or `test` element has one or more `class` child elements specifying the class or classes to which the component or test belongs. The content of each class element is a class name. The component or test element may also have zero or more additional child elements describing its properties. Each property is expressed as a child element whose tag name is a property name and whose content is the the value of the property. For convenience, agents can abbreviate the property name using XML namespaces. The order of the classes and properties does not matter. The value of descriptive properties is expressed as a string. Leading and trailing whitespace is ignored. The value of relationship properties is expressed as a `componentRef` child element. A `componentRef` element has an attribute `ref` whose value is the `id` value of another component in the component graph. The metadata in the `observation` element applies to all properties in the observation. Below I give an example of an observation of an outbound connectivity test.

```
<observation id="obs-3"
             xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
             time="1168356309119"
             originator="http://18.26.0.100/userAgent"
             cached="false"
```

```
            expires="1170451641706">
  <test id="test-6">
    <class>
      http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test
    </class>
    <com:srcIP>       127.0.0.1                   </com:srcIP>
    <com:probeURI>    capri.csail.mit.edu         </com:probeURI>
    <com:probeResult> OK                          </com:probeResult>
    <com:localNet>    <componentRef ref="com-2"/> </com:localNet>
  </test>
</observation>
```

Representing beliefs

Agents represent beliefs and likelihoods using the `belief` element. The metadata attributes of a `belief` element are identical to those of an `observation` element. A belief has four child elements: a `subject` indicating the component for which the belief applies, a `property` specifying the property of the component the belief describes, a `fromEvidence` element listing the evidence used to produce the belief, and either a `distribution` indicating the probability of each of the possible values of the property or a `likelihood` element indicating the probability of the evidence given each possible value of the property. The `subject` element has one child, either a `component`, `test`, or `componentRef` element. The format of each of these elements is described above in Section A.2.1. The content of the `property` element is the name of the property associated with the belief. The `fromEvidence` element contains zero or more `evidence` child nodes describing the evidence used to infer the belief. If the `fromEvidence` element has no `evidence` child nodes, then the belief was inferred using only probabilistic dependency knowledge without any additional evidence. Each `evidence` node has a `propPath` attribute indicating the property path to a property for which the creator of this belief had evidence it used to infer the belief. The `distribution` element has two or more `entry` child elements. Each `entry` element has a `value` attribute and a `p` attribute such that $P(\textit{property} = \textit{value} | \textit{evidence}) = p$. A `likelihood` element has the same format, except that for the `entry` elements for a `likelihood`, $P(\textit{evidence} | \textit{property} = \textit{value})$. Below I give an example of a belief about the status of an *HTTP Server* component inferred from the number of consecutive failures to the server.

```
<belief id="bel-2"
        xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
        originator="http://18.26.0.100:8111/da"
        time="1160577157224"
        cached="true"
        expires="1170451641706">
  <subject>
    <component id="com-1">
      <class>
         http://capri.csail.mit.edu/2006/capri/common#HTTP_Server
      </class>
```

```
      <com:hostHash>+0THLqaZnxk9I8bU5ZgDGA==</com:hostHash>
    </component>
  </subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <distribution>
    <entry value="OK"   p="0.8"/>
    <entry value="FAIL" p="0.2"/>
  </distribution>
  <fromEvidence>
    <evidence
propPath="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
    />
  </fromEvidence>
</belief>
```

Representing dependency knowledge

Agents represent dependency knowledge using the `knowledge` element. Knowledge has the same five metadata attributes as observations and beliefs. A `knowledge` element has three child elements. The `subject` and `property` child elements contain the names of the component class and property, respectively, for which the dependency knowledge applies. The `cpt` child specifies the conditional probability table associated with the knowledge. A `cpt` element has a single `parents` child element containing a whitespace delimited list of parent property paths. The order of the parents matters. A `cpt` element also has two or more `entry` child nodes. Each `entry` child node has three attributes `value`, `parentVals`, and `p` such that $P(\textit{property} = \textit{value} | \texttt{parents} = \texttt{parentVals}) = p$. The `parentVals` attribute is a whitespace delimited list of parent property values in the order specified in the `parents` element. Below I give an example of a piece of dependency knowledge for the `asHopTestResult` property of the *AS Hop Test* component class.

```
<knowledge id="knowledge-2"
           originator="http://18.26.0.100:8111/da"
           time="1160577157224"
           cached="false"
           expires="1170451641706">
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#asHopTestResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asHop|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.995000' parentVals='FAIL' value='FAIL'/>
    <entry p='0.020000' parentVals='OK'   value='FAIL'/>
```

```
    <entry p='0.005000' parentVals='FAIL' value='OK'/>
    <entry p='0.980000' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
```

A.2.2 Representing diagnostic messages

Diagnostic agents communicate diagnostic messages in batches. A batch can contain one or more diagnostic messages. Agents represent a batch using a `capriMessage` element. A `capriMessage` element contains a `version` attribute specifying the DMEP protocol version, a `clientAgent` attribute describing the implementation of the agent sending the batch, a `clientVer` attribute indicating the version number of the sending agent, and a `time` attribute indicating the time at which the batch is sent.

A `capriMessage` element contains one or more child elements, where each child element is either an `observationRequest`, `beliefRequest`, `knowledgeRequest`, `diagRequest`, `notification`, `observationResponse`, `beliefResponse`, `knowledgeResponse`, `diagResponse`, `notificationResponse`, or `error`.

Observation requests

An observation request is described using an `observationRequest` element. The `observationRequest` element contains a `requester` attribute indicating the URI of the requesting agent, a `requestID` attribute, a `serviceID` attribute indicating the observation service corresponding to the request, and an `expires` attribute indicating the expiration time of this request in milliseconds since Jan 1, 1970 UTC.

An `observationRequest` element may optionally have an `inputComponent` child element specifying the input component for the service requested. An `inputComponent` element has exactly one child element, either a `component`, `test`, or `componentRef` as described in Section A.2.1. In addition, an `observationRequest` element may have an optional `body` child element describing additional diagnostic information to communicate. A `body` element contains zero or more `observation`, `belief`, and `knowledge` elements. For example, a request for the number of consecutive failures to a web server may look as follows.

```
<observationRequest
  xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
  requester="http://18.26.0.100/regionalAgent"
  requestID="101"
  serviceID="obs:webserver.cfts"
  expires="1160577160000">
<inputComponent>
  <component>
    <class>
      http://capri.csail.mit.edu/2006/capri/common#HTTP_Server
    </class>
    <com:ipAddr>   192.168.10.10            </com:ipAddr>
```

```
    <com:hostHash> +0THLqaZnxk9I8bU5ZgDGA== </com:hostHash>
    <com:connTime> 1167766762747            </com:connTime>
  </component>
</inputComponent>
</observationRequest>
```

Belief requests

Agents express belief requests using the `beliefRequest` element. Belief requests have the same form as observation requests, except that a `beliefRequest` element also has a `budget` attribute. Below I give an example of a belief request for *HTTP Server* `status`.

```
<beliefRequest
  xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
  requester="http://18.26.0.100/regionalAgent"
  requestID="102"
  serviceID="bel:webserver.status"
  budget="1000"
  expires="1160577160000">
<inputComponent>
  <component id="com-1">
    <class>
      http://capri.csail.mit.edu/2006/capri/common#HTTP_Server
    </class>
    <com:hostHash> +0THLqaZnxk9I8bU5ZgDGA== </com:hostHash>
    <com:connTime> 1167766762747            </com:connTime>
  </component>
</inputComponent>
</beliefRequest>
```

Knowledge requests

Agents express knowledge requests using the `knowledgeRequest` element. Knowledge requests have the same form as observation and belief requests, except that they do not require an `inputComponent`. For example,

```
<knowledgeRequest
  requester="http://18.26.0.100/regionalAgent"
  requestID="103"
  serviceID="knowledge:http"
  expires="1160577160000" />
```

Diagnosis requests

Agents express diagnosis requests using the `diagRequest` element. A `diagRequest` has the same format as a `beliefRequest`, except that a `diagRequest` element also has a `explanations` child element specifying the set of alternative explanations for the failure. An `explanations` element has two or more `explanation` child elements. Each

`explanation` child element contains one or more `componentRef` elements. Each `componentRef` element has a `ref` attribute indicating a component, and a `status` element specifying the status of the component. Each component in each explanation must be in the list of candidate explanations for the service corresponding to the diagnosis request. Part of an *HTTP Connection* failure diagnosis request is given below.

```
<diagRequest requester="http://18.26.0.100:8111/userAgent"
             requestID="501"
             serviceID="diag:http"
             budget="1000"
             expires="1160577160000"
             confThresh="0.9">
<inputComponent>
  <componentRef ref="com-1" />
</inputComponent>
<explanations>
  <explanation>
    <componentRef ref="com-2" status="FAIL" />
  </explanation>
  <explanation>
    <componentRef ref="com-3" status="FAIL" />
  </explanation>
  <explanation>
    <componentRef ref="com-4" status="FAIL" />
  </explanation>
  <explanation>
    <componentRef ref="com-5" status="FAIL" />
  </explanation>
</explanations>
<body>
<observation time="1168356307879">
  <component id="com-1">
    <class>
      http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
    </class>
    <com:srcIP>            127.0.0.1                  </com:srcIP>
    <com:destHash>         aRexNPkGuP1p9s54cXAIQg==   </com:destHash>
    <com:connTime>         1168356307879              </com:connTime>
    <core:status>          FAIL                       </core:status>

    <com:dnsLookup>        <componentRef ref="com-4"/> </com:dnsLookup>
    <com:ipRouting>        <componentRef ref="com-5"/> </com:ipRouting>
    <com:httpServer>       <componentRef ref="com-3"/> </com:httpServer>
    <com:localNet>         <componentRef ref="com-2"/> </com:localNet>
    <com:outboundConnTest> <componentRef ref="com-6"/> </com:outboundConnTest>
    <com:ffoxErrorTest>    <componentRef ref="com-7"/> </com:ffoxErrorTest>
  </component>
</observation>
```

```
<observation time="1168356309119">
  <component id="com-2">
    <class>
      http://capri.csail.mit.edu/2006/capri/common#Local_Network
    </class>
    <com:ipAddr>                  127.0.0.1 </com:ipAddr>
    <com:consecFailuresFromUser> 0         </com:consecFailuresFromUser>
  </component>
</observation>
...
</body>
```

Note several characteristics of this diagnostic request. It contains a set of observations relating to the failure (a failure story). Each observation contains a description of a component individual, including the properties of the component such as its identity, its parent (dependent) components, its status, and other characteristics. The properties a component may have are defined in the component class ontology. An observation also includes some metadata, including the time the observation was made and where the the observation came from. Also, the component IDs above are only used to construct the graph structure and do not mean anything outside this failure story. Each explanation is a status assignment to one or more components. The task of diagnosis is to identify the likelihood of each explanation.

Notifications

Agents express notifications using the `notification` element. A notification element has only two attributes, a `serviceID` specifying the notification subscription for this notification, and a `notificationID`. The content of a `notification` element is the same as an observation or belief request, and can contain `inputComponent` and a `body` child elements. Below is an example of an *HTTP Connection* notification.

```
<notification
  xmlns:core="http://capri.csail.mit.edu/2006/capri/core#"
  xmlns:com="http://capri.csail.mit.edu/2006/capri/common#"
  serviceID="notify:connHist"
  notificationID="10">
<inputComponent>
  <componentRef ref="com-1" />
</inputComponent>
<body>
  <observation time="1160577160000">
    <component id="com-1">
      <class>
        http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
      </class>
      <com:destHash>    +0THLqaZnxk9I8bU5ZgDGA== </com:destHash>
      <com:connTime>    1160577158287            </com:connTime>
      <core:status>     OK                       </core:status>
```

```
      <com:elapsedTime> 340                    </com:elapsedTime>
      <com:srcIP>       1.2.3.4                </com:srcIP>
      <com:destIP>      50.60.70.80            </com:destIP>
    </component>
  </observation>
</body>
</notification>
```

Observation response

Agents express observation responses using the `observationResponse` element. An `observationResponse` contains the `requestID` of the corresponding `observationRequest`, the corresponding `serviceID`, and a `responseStatus`. The value of `responseStatus` is either 0 to indicate an error producing the response, or 1 indicating a successful response. An `observationResponse` element contains a `body` child element with zero or more `observation`, `belief`, and `knowledge` child elements. Note that an observation response may contain observations, beliefs, and dependency knowledge not specified in the corresponding service description so that an agent can propagate additional information to the requester even when not explicitly requested. Also note that `componentRef` elements in a response can refer to components described in the request.

Below I give an example of an observation response for a request for server statistics may look as follows:

```
<observationResponse
  xmlns:ss="http://capri.csail.mit.edu/2006/capri/servstats#"
  responseStatus="1"
  serviceID="obs:stats"
  requestID="101"
  time="1176087435536">
<body>
  <observation
    id="obs-1"
    time="1176087435532"
    originator="http://127.0.0.1:80/statsAgent"
    cached="false"
    expires="1176087435532">

    <component id="http://127.0.0.1:80/statsAgent/com-1">
      <class>
        http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
      </class>
      <ss:users>            3                        </ss:users>
      <ss:destHash>         +0THLqaZnxk9I8bU5ZgDGA== </ss:destHash>
      <ss:avgLatency>       336                      </ss:avgLatency>
      <ss:recentStatusDist> 0,2|1,2|14,3             </ss:recentStatusDist>
      <ss:lastFailure>      250000                   </ss:lastFailure>
      <ss:lastSuccess>      12300                    </ss:lastSuccess>
```

```
    </component>
  </observation>
</body>
</observationResponse>
```

Belief Response

Agents express belief responses using the `beliefResponse` element. A `beliefResponse` element has the same format as an `observationResponse`, though the information provided in the body may differ.

KnowledgeResponse

Agents express knowledge responses using the `knowledgeResponse` element. A `knowledgeResponse` element has the same format as an `observationResponse`, though the information provided in the body may differ.

Diagnosis Response

Agents express diagnosis responses using the `diagResponse` element. A diagnosis response has the same format as an `observationResponse`, except that it also contains a `explanations` child element listing the probability of each candidate explanation provided in the request. The `explanations` element contains one or more `explanation` elements. The content of the `explanation` element is the same as in a `diagRequest`. Each `explanation` element has an attribute `p` indicating the probability all the components in the explanation have the specified status values. To provide additional information to requesters about the beliefs and evidence used to produce the diagnosis, an agent may also include additional observations and beliefs in the body of the response. Below is an example of a diagnostic response for an HTTP Connection failure diagnosis.

```
<diagResponse
  responseStatus="1"
  serviceID="diag:http"
  requestID="321"
  time="1176088430691">
  <explanations>
    <explanation p="1.0000">
      <componentRef status="FAIL" ref="com-4"/>
    </explanation>
    <explanation p="0.0431">
      <componentRef status="FAIL" ref="com-3"/>
    </explanation>
    <explanation p="0.0016">
      <componentRef status="FAIL" ref="com-5"/>
    </explanation>
    <explanation p="0.0000">
      <componentRef status="FAIL" ref="com-2"/>
```

```
  </explanation>
</explanations>
<body>
  <belief
    id="bel-1"
    time="1176088430689"
    originator="http://142.150.238.12:8111/regionalAgent"
    cached="false"
    expires="1176092030689">
    <subject>
      <componentRef ref="com-4"/>
    </subject>
    <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
    <distribution>
      <entry p="0.999997" value="FAIL"/>
      <entry p="0.000003" value="OK"/>
    </distribution>
    <fromEvidence/>
  </belief>
  <belief
    id="bel-2"
    time="1176088430689"
    originator="http://142.150.238.12:8111/regionalAgent"
    cached="false"
    expires="1176092030689">
    <subject>
      <componentRef ref="com-3"/>
    </subject>
    <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
    <distribution>
      <entry p="0.043079" value="FAIL"/>
      <entry p="0.956921" value="OK"/>
    </distribution>
    <fromEvidence/>
  </belief>
  <belief
    id="bel-3"
    time="1176088430689"
    originator="http://142.150.238.12:8111/regionalAgent"
    cached="false"
    expires="1176092030689">
    <subject>
      <componentRef ref="com-5"/>
    </subject>
    <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
    <distribution>
      <entry p="0.001625" value="FAIL"/>
```

```
        <entry p="0.998375" value="OK"/>
      </distribution>
      <fromEvidence/>
    </belief>
    <belief
      id="bel-4"
      time="1176088430689"
      originator="http://142.150.238.12:8111/regionalAgent"
      cached="false"
      expires="1176092030689">
      <subject>
        <componentRef ref="com-2"/>
      </subject>
      <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
      <distribution>
        <entry p="0.000005" value="FAIL"/>
        <entry p="0.999995" value="OK"/>
      </distribution>
      <fromEvidence/>
    </belief>
  </body>
</diagResponse>
```

Notification response

An agent that receives a notification responds with a notification response to acknowledge receipt of the notification. A `notificationResponse` element contains only a `notificationID` attribute and a `responseStatus` attribute.

Error response

An agent that cannot process a request produces an error response using a `error` element. The content of the `error` element is a text string describing the error.

A.3 Service description language

This section describes the XML service description language that CAPRI agents use to represent their diagnostic capabilities. See Appendix B.2 for actual service descriptions used in the prototype implementation described in Chapter 8.

A.3.1 Directory updates

Agents communicate service descriptions to a service directory using directory update messages. A directory update message has the root element `directoryUpdate`. A `directoryUpdate` element contains one or more `serviceAdvertisement` elements. Each `serviceAdvertisement` element contains a `serviceID` attribute; a `time` attribute indicating when

the service was defined; a messageType attribute whose value is either observationRequest, beliefRequest, diagRequest, or notification; an agentURI attribute indicating the URI of the agent providing the service; a cost attribute; an optional requesterType attribute specifying the types of requesters that may use this service; an optional requesterASRange attribute specifying a comma delimited list of Internet autonomous system numbers of requesters allowed to use this service, and an optional inputClass attribute. If the requesterType attribute is not present, then all types of requesters are allowed. If the requesterASRange attributes is not present, then requesters from all ASes are allowed. A service advertisement with the same agentURI and serviceID as another service but with a greater time value supersedes the older service.

A serviceAdvertisement may have zero or more inputProperty, outputObservation, outputBelief, outputKnowledge, and candidateExplanation child elements.

An inputProperty element has a propPath attribute specifying the property path of the input property relative to the input component. An inputProperty also has an optional required property indicating whether the property is required to use the service. By default a property is not required. An input property with input restrictions also has indexFunc and indexRange attributes. See Section 5.1.3 for the list of allowable index functions.

An observation, belief, or diagnosis service may have one or more output observations. An outputObservation has a propPath attribute specifying the property path of an observation produced by the service relative to the input component.

A belief or diagnosis service may have one or more output beliefs. An outputBelief also has a propPath attribute referring to the component and property of the belief the service produces. In addition, an outputBelief also has zero or more fromEvidence child elements. Each fromEvidence child element has a propPath attribute indicating a property that the provider agent can access as evidence to produce the belief.

An observation, belief, diagnosis, or knowledge service may have one or more knowledge outputs. An outputKnowledge element has a subject attribute and a property attribute indicating the corresponding subject and property for a piece of knowledge the service produces. In addition, an outputKnowledge element has zero or more parent child elements. Each parent element has a propPath attribute indicating the path to one of the parent properties for the dependency knowledge.

A diagnosis service may provide one or more candidate explanations. A candidateExplanation element has a propPath attribute indicating the path to the property for the candidate explanation.

A directory service responds to a directory update with a directory update response. A directory update response contains a single directoryUpdateResponse element with a single attribute responseStatus, whose value is 1 if the directory update was processed successfully or 0 otherwise.

A.3.2 Service map requests

An agent requests the list of available services using an service map request. A service map request is an XML message with root element serviceMapRequest. A serviceMapRequest element has a version attribute indicating the service directory proto-

col version, a `clientVer` attribute indicating the version of the requesting agent, and an `agentType` attribute indicating the type of agent (e.g. user, specialist, or regional).

A.3.3 Service map responses

A directory service responds to a service map request with a service map response. A service map response is an XML message with root element `serviceMap`. A `serviceMap` element contains zero or more `serviceAdvertisement` child nodes. If an agent has a choice between two services with the same utility, the agent should use the service that appears first in the service map. This allows the directory service to balance load and control the distribution of requests across multiple agents.

A.3.4 Service retractions

An agent may also retract a service that it no longer offers. A retraction message consists of a `directoryUpdate` element with `serviceID`, `agentURI`, and `time` attributes corresponding to the service to retract. The `directoryUpdate` element for a retraction message also has an additional attribute `retract` having the value `true`. After the directory service receives a retraction message for a service, it no longer advertises the service to other agents.

Appendix B

Prototype Implementation Details

This appendix contains the ontology, service descriptions, and dependency knowledge used in the prototype implementation described in Chapter 8.

B.1 Component class and property definitions

In the prototype implementation described in Chapter 8, agents load component class and property definitions from the core component ontology, the common component ontology, a ServStats component ontology, and a CoDNS component ontology. These ontologies are provided below, and can be downloaded using their respective URIs.

B.1.1 Core component ontology

```
<componentOntology>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/core#status">
  <range>
    <entry value="OK" />
    <entry value="FAIL" />
  </range>
</descriptiveProperty>

</componentOntology>
```

B.1.2 Common component ontology

```
<componentOntology>

<!-- ********************** Classes ************************** -->

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server">
```

```
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#ipAddr" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destHash" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#connTime" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destHash" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#Local_Network">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#ipAddr" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#hostname" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#probeURI" />
</componentClass>

<componentClass
    name="http://capri.csail.mit.edu/2006/capri/common#IP_Routing">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destIP" />
</componentClass>

<componentClass
```

```
  name="http://capri.csail.mit.edu/2006/capri/common#Verify_DNS_Lookup_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#hostname" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#AS_Path">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcAS" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destAS" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#AS_Path_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcAS" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destAS" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#AS_Hop">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcAS" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destAS" />
</componentClass>

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#srcAS" />
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#destAS" />
</componentClass>

<!-- ********************** Properties ************************* -->

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#srcAS" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#destAS" />

<!-- a comma separated list of ASes along the path -->
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#pathASes" />
```

```
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asHopTestResult">
  <range>
    <entry value="OK" />
    <entry value="FAIL" />
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asPathTestResult">
  <range>
    <entry value="OK" />
    <entry value="FAIL" />
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#connTime" />

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser">
  <range>
    <entry value="0" />
    <entry value="1" />
    <entry value="2" />
    <entry value="3" />
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer">
  <range>
    <entry value="0" />
    <entry value="1" />
    <entry value="2" />
    <entry value="3" />
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#destHash" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#destIP" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#probeURI" />
```

```
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult">
  <range>
    <entry value="LOOKUP_ERROR_CONFIRMED" />
    <entry value="LOOKUP_ERROR_UNCONFIRMED" />
    <entry value="CORRECT" />
    <entry value="INCORRECT" />
    <entry value="LOOKUP_ERROR" />
    <entry value="ALIAS" />
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode">
  <range>
    <entry value="0" />    <!-- Success -->
    <entry value="2" />    <!-- Canceled by user -->
    <entry value="13" />   <!-- Connection refused -->
    <entry value="14" />   <!-- Connection timed out -->
    <entry value="20" />   <!-- Connection reset -->
    <entry value="21" />   <!-- FTP login failed -->
    <entry value="22" />   <!-- FTP CWD failed -->
    <entry value="30" />   <!-- Server not found -->
    <entry value="42" />   <!-- HTTP proxy not found -->
    <entry value="71" />   <!-- Net interrupt -->
    <entry value="72" />   <!-- HTTP proxy connection refused -->
  </range>
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#hostHash" />

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#hostname" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#ipAddr" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#ipAddrs" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#probeResult">
  <range>
    <entry value="OK" />
    <entry value="FAIL" />
  </range>
```

```
</descriptiveProperty>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/common#srcIP" />

<descriptiveProperty
name="http://capri.csail.mit.edu/2006/capri/common#totalConsecFailuresFromUser"
/>
<descriptiveProperty
name="http://capri.csail.mit.edu/2006/capri/common#totalConsecFailuresToServer"
/>

<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asPath"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Path"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#ipRouting"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#nextASPath"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Path"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asHop"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Hop"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#nextASHop"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Hop"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asPathTest"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Path_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#asPath"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#asHopTest"
  range="http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#asHop"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#dnsLookup"
  range="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#httpServer"
  range="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
/>
```

```
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#ipRouting"
  range="http://capri.csail.mit.edu/2006/capri/common#IP_Routing"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#localNet"
  range="http://capri.csail.mit.edu/2006/capri/common#Local_Network"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#verifyDNSLookupTest"
  range="http://capri.csail.mit.edu/2006/capri/common#Verify_DNS_Lookup_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#dnsLookup"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest"
  range="http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#httpConn"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest"
  range="http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#httpConn"
/>
<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/common#httpConn"
  range="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection"
/>

</componentOntology>
```

B.1.3 ServStats ontology

```
<componentOntology>

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#users" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#rootCause" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#consecFailures" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#avgLatency" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#lastSuccess" />
<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/servstats#lastFailure" />
<descriptiveProperty
```

```
  name="http://capri.csail.mit.edu/2006/capri/servstats#recentStatusDist" />

</componentOntology>
```

B.1.4 CoDNS ontology

```
<componentOntology>
<!-- ************************ Classes **************************** -->

<componentClass
  name="http://capri.csail.mit.edu/2006/capri/planetlab#CoDNS_Lookup_Test">
  <idProperty
    name="http://capri.csail.mit.edu/2006/capri/common#hostname" />
</componentClass>

<!-- ************************ Properties ************************ -->

<descriptiveProperty
  name="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult">
  <range>
    <entry value="LOOKUP_ERROR_CONFIRMED" />
    <entry value="LOOKUP_ERROR_UNCONFIRMED" />
    <entry value="CORRECT" />
    <entry value="INCORRECT" />
    <entry value="LOOKUP_ERROR" />
  </range>
</descriptiveProperty>

<relationshipProperty
  name="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupTest"
  range="http://capri.csail.mit.edu/2006/capri/planetlab#CoDNS_Lookup_Test"
  symmetricProp="http://capri.csail.mit.edu/2006/capri/common#dnsLookup"
/>

</componentOntology>
```

B.2 Service descriptions

In each of the service descriptions below, agents advertising the services replace the string `localhost` with their own IP address and `localAS` with their own AS number.

B.2.1 Regional agent

```
<directoryUpdate>

<!-- ************************ For local users ********************** -->
```

```
<!-- Diagnostic requests from local agents in same AS.  "localAS" will
get replaced with the local AS number.  Since this one is lower cost
than the general one, users within the same AS as this regional agent
will prefer this one.  -->

<serviceAdvertisement
  serviceID="diag:http(localAS)"
  time="1172597022"
  messageType="diagRequest"
  agentURI="http://localhost/regionalAgent"
  cost="1000"
  requesterType="user"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status"
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/common#srcIP"
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/common#destIP"
    required="true"
  />

  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
```

```
              http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>

<!-- This is for connection history notifications, which will get
forwarded to history and stats agents. -->

<serviceAdvertisement
  serviceID="notify:connHist(localAS)"
  time="1172597022"
  messageType="notification"
  agentURI="http://localhost/regionalAgent"
  cost="1000"
  requesterType="user"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection" >

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
</serviceAdvertisement>

<!-- Diag history notifications that get propagated to the learning agent. -->

<serviceAdvertisement
  serviceID="notify:diagHist(localAS)"
  time="1172597022"
  messageType="notification"
  agentURI="http://localhost/regionalAgent"
  cost="1000"
  requesterType="user"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
```

```
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#localNet" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
            http://capri.csail.mit.edu/2006/capri/core#status" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
            http://capri.csail.mit.edu/2006/capri/common#ipAddr" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser"
/>
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
    http://capri.csail.mit.edu/2006/capri/common#totalConsecFailuresFromUser"
/>
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
            http://capri.csail.mit.edu/2006/capri/core#status" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
/>
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
            http://capri.csail.mit.edu/2006/capri/common#hostname" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
            http://capri.csail.mit.edu/2006/capri/core#status" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
            http://capri.csail.mit.edu/2006/capri/core#status" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
            http://capri.csail.mit.edu/2006/capri/common#srcIP" />
<inputProperty
  propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
            http://capri.csail.mit.edu/2006/capri/common#destIP" />
<inputProperty
```

```
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest|
              http://capri.csail.mit.edu/2006/capri/common#probeResult" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest|
              http://capri.csail.mit.edu/2006/capri/common#probeURI" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
</serviceAdvertisement>

<!-- To support stats.  Recursively request from appropriate stats agent. -->
<serviceAdvertisement
  serviceID="obs:stats(localAS)"
  time="1172597022"
  messageType="observationRequest"
  agentURI="http://localhost/regionalAgent"
  cost="1000"
  requesterType="user"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash"
    required="true" />

  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#users" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#rootCause" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#consecFailures"
  />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#avgLatency" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastSuccess" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastFailure" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#recentStatusDist"
  />
</serviceAdvertisement>

<!-- Diag knowledge for local agents -->
```

```
<serviceAdvertisement
  serviceID="knowledge:http(localAS)"
  time="1172597022"
  messageType="knowledgeRequest"
  agentURI="http://localhost/regionalAgent"
  cost="100"
  requesterType="user"
  requesterASRange="localAS">

  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Local_Network"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Local_Network"
property="http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser"
  >
     <parent propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection"
     property="http://capri.csail.mit.edu/2006/capri/core#status">
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#IP_Routing"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
property="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
  >
```

```
      <parent propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test"
     property="http://capri.csail.mit.edu/2006/capri/common#probeResult" >
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test"
     property="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" >
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                 http://capri.csail.mit.edu/2006/capri/common#localNet|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                 http://capri.csail.mit.edu/2006/capri/common#httpServer|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                 http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                 http://capri.csail.mit.edu/2006/capri/common#ipRouting|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
   </outputKnowledge>
</serviceAdvertisement>


<!-- ************************* For all users ************************* -->


<!-- service for non AS users -->
<serviceAdvertisement
  serviceID="diag:http"
  time="1172597022"
  messageType="diagRequest"
  agentURI="http://localhost/regionalAgent"
  cost="2000"
  requesterType="user"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status"
```

```
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/common#srcIP"
    required="false"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/common#destIP"
    required="true"
  />

  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <candidateExplanation
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>

<!-- This is for connection history notifications, which will get
forwarded to history and stats agents. -->
<serviceAdvertisement
  serviceID="notify:connHist"
  time="1172597022"
  messageType="notification"
  agentURI="http://localhost/regionalAgent"
  cost="2000"
  requesterType="user"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection"
>
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash" />
```

```
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
</serviceAdvertisement>

<!-- Diag history notifications that get propagated to the learning agent. -->
<serviceAdvertisement
  serviceID="notify:diagHist"
  time="1172597022"
  messageType="notification"
  agentURI="http://localhost/regionalAgent"
  cost="2000"
  requesterType="user"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection" >
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
              http://capri.csail.mit.edu/2006/capri/common#ipAddr" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/common#totalConsecFailuresFromUser"
  />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
```

```
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
      http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/common#hostname" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
              http://capri.csail.mit.edu/2006/capri/common#destIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest|
              http://capri.csail.mit.edu/2006/capri/common#probeResult" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#outboundConnTest|
              http://capri.csail.mit.edu/2006/capri/common#probeURI" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
</serviceAdvertisement>

<!-- To support stats.  Recursively request from appropriate stats agent. -->
<serviceAdvertisement
  serviceID="obs:stats"
  time="1172597022"
  messageType="observationRequest"
  agentURI="http://localhost/regionalAgent"
  cost="2000"
  requesterType="user"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash"
```

```
    required="true" />

  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#users" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#rootCause" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#consecFailures"/>
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#avgLatency" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastSuccess" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastFailure" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#recentStatusDist"
  />
</serviceAdvertisement>

<!-- Diag knowledge for user agents -->
<serviceAdvertisement
  serviceID="knowledge:http"
  time="1172597022"
  messageType="knowledgeRequest"
  agentURI="http://localhost/regionalAgent"
  cost="200"
  requesterType="user"
>

  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Local_Network"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#Local_Network"
property="http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser"
  >
     <parent propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection"
     property="http://capri.csail.mit.edu/2006/capri/core#status">
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#httpServer|
                 http://capri.csail.mit.edu/2006/capri/core#status" />
```

```
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#ipRouting|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
   </outputKnowledge>
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
      property="http://capri.csail.mit.edu/2006/capri/core#status" />
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#IP_Routing"
      property="http://capri.csail.mit.edu/2006/capri/core#status" />
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
      property="http://capri.csail.mit.edu/2006/capri/core#status" />
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
property="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
   >
      <parent propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
   </outputKnowledge>
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test"
      property="http://capri.csail.mit.edu/2006/capri/common#probeResult" >
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#localNet|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
   </outputKnowledge>
   <outputKnowledge
      subject="http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test"
      property="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" >
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                  http://capri.csail.mit.edu/2006/capri/common#localNet|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                  http://capri.csail.mit.edu/2006/capri/common#httpServer|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                  http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
                  http://capri.csail.mit.edu/2006/capri/core#status" />
      <parent
        propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
                  http://capri.csail.mit.edu/2006/capri/common#ipRouting|
```

```
                http://capri.csail.mit.edu/2006/capri/core#status" />
   </outputKnowledge>
</serviceAdvertisement>

<!-- ************************* Local actions ************************ -->

<!-- Look up the AS path for an IP Routing component to incorporate cached
     AS Path and AS Hop information. -->
<serviceAdvertisement
  serviceID="aspath"
  time="1170697052"
  messageType="observationRequest"
  agentURI="http://localhost/regionalAgent"
  cost="10"
  requesterType="local"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#IP_Routing" >

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP"
    required="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP"
    required="true" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/common#asPath" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/common#asPath|
              http://capri.csail.mit.edu/2006/capri/common#srcAS" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/common#asPath|
              http://capri.csail.mit.edu/2006/capri/common#destAS" />
</serviceAdvertisement>

</directoryUpdate>
```

B.2.2 Web server history test agent

```
<directoryUpdate>

<!-- %INDEX_RANGE gets replaced with either 0, 1, 2, or 3 depending on the web
     server history agent -->
<serviceAdvertisement
  serviceID="bel:webserver.status"
  time="1170545800"
  messageType="beliefRequest"
  agentURI="http://localhost/historyAgent"
  cost="1000"
```

```
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddr"
    required="true"
    indexFunc="iptoint,mod 4"
    indexRange="%INDEX_RANGE" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime"
    required="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostHash" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#destHash" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#destIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#connTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#httpConn|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
propPath="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
    />
  </outputBelief>
```

```
  <outputObservation
propPath="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
  />

  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>

<serviceAdvertisement
  serviceID="obs:webserver.cfts"
  time="1170545800"
  messageType="observationRequest"
  agentURI="http://localhost/historyAgent"
  cost="1100"
  requesterType="learning"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Server">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddr"
    required="true"
    indexFunc="iptoint,mod 4"
    indexRange="%INDEX_RANGE" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostHash" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime"
    required="true" />
  <outputObservation
propPath="http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer"
  />
  <outputObservation
propPath="http://capri.csail.mit.edu/2006/capri/common#totalConsecFailuresToServer"
  />
</serviceAdvertisement>

<serviceAdvertisement
  serviceID="notify:connHist(destIP)"
  time="1170545800"
  messageType="notification"
  agentURI="http://localhost/historyAgent"
  cost="1000"
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">

  <inputProperty
```

```
    propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP"
    required="true"
    indexFunc="iptoint,mod 4"
    indexRange="%INDEX_RANGE" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
</serviceAdvertisement>
</directoryUpdate>
```

B.2.3 DNS lookup test agent

```
<directoryUpdate>

<!-- Prefer requesters in the same AS by offering a lower cost. -->
<serviceAdvertisement
  serviceID="bel:dnslookup.status(local)"
  time="1170701032"
  messageType="beliefRequest"
  agentURI="http://localhost/dnsAgent"
  cost="9000"
  requesterType="regional"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddrs"
    required="true" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
propPath="http://capri.csail.mit.edu/2006/capri/common#verifyDNSLookupTest|
          http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult"
```

```
    />
  </outputBelief>
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/common#verifyDNSLookupTest|
              http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>

<!-- The ad for non local AS requesters. ******************************  -->

<serviceAdvertisement
  serviceID="bel:dnslookup.status"
  time="1170701032"
  messageType="beliefRequest"
  agentURI="http://localhost/dnsAgent"
  cost="10000"
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddrs"
    required="true" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
    propPath="http://capri.csail.mit.edu/2006/capri/common#verifyDNSLookupTest|
              http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult" />
  </outputBelief>
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/common#verifyDNSLookupTest|
              http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult" />
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>
</directoryUpdate>
```

B.2.4 AS path test agent

```
<directoryUpdate>
<serviceAdvertisement
```

```
  serviceID="bel:iprouting.status"
  messageType="beliefRequest"
  agentURI="http://localhost/ipAgent"
  time="1170545800"
  cost="100000"
  requesterType="regional"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#IP_Routing"
  indexRange="0">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP"
    required="true"
    indexFunc="asn"
    indexRange="localAS" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP"
    required="true" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
      propPath="http://capri.csail.mit.edu/2006/capri/common#asPath|
                http://capri.csail.mit.edu/2006/capri/common#asPathTest|
                http://capri.csail.mit.edu/2006/capri/common#asPathTestResult"
    />
  </outputBelief>
  <outputKnowledge
     subject="http://capri.csail.mit.edu/2006/capri/common#IP_Routing"
     property="http://capri.csail.mit.edu/2006/capri/core#status" />
</serviceAdvertisement>
</directoryUpdate>
```

B.2.5 Stats agent

```
<directoryUpdate>
<!-- %INDEX_RANGE gets replaced with either 0, 1, 2, or 3 depending on the stats
     agent -->
<serviceAdvertisement
  serviceID="notify:connHist(destHash)"
  time="1170545800"
  messageType="notification"
  agentURI="http://localhost/statsAgent"
  cost="100"
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection"
>
```

```
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/core#status" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash"
    required="true"
    indexFunc="b64toint,mod 4"
    indexRange="%INDEX_RANGE" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#srcIP" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#connTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#elapsedTime" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ffoxErrorTest|
              http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode" />
</serviceAdvertisement>

<serviceAdvertisement
  serviceID="obs:stats"
  time="1170545799"
  messageType="observationRequest"
  agentURI="http://localhost/statsAgent"
  cost="100"
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection">
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#destHash"
    required="true"
    indexFunc="b64toint,mod %NUM_INDEXES"
    indexRange="%INDEX_RANGE" />

  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#users" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#rootCause" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#consecFailures"/>
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#avgLatency" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastSuccess" />
  <outputObservation
```

```
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#lastFailure" />
  <outputObservation
    propPath="http://capri.csail.mit.edu/2006/capri/servstats#recentStatusDist"
  />
</serviceAdvertisement>
</directoryUpdate>
```

B.2.6 CoDNS lookup test agent

```
<directoryUpdate>

<!-- Prefer requesters in the same AS by offering a lower cost.  Have
the same ad, but one for agents in the same AS.  CoDNS is cheaper than
regular DNS lookup. -->

<serviceAdvertisement
  serviceID="bel:dnslookup.status(local)[codns]"
  time="1170701032"
  messageType="beliefRequest"
  agentURI="http://localhost/codnsAgent"
  cost="400"
  requesterType="regional"
  requesterASRange="localAS"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup"
>
<!-- here the indexrange is simply computed from the hostname -->
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="true"
    index="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddrs"
    required="true" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
  propPath="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupTest|
            http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult"
    />
  </outputBelief>
  <outputObservation
  propPath="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupTest|
            http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult"
  />
</serviceAdvertisement>
```

```
<!-- ******* The ad for non local AS requesters. ***************  -->

<serviceAdvertisement
  serviceID="bel:dnslookup.status[codns]"
  time="1170701032"
  messageType="beliefRequest"
  agentURI="http://localhost/codnsAgent"
  cost="500"
  requesterType="regional"
  inputClass="http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup">

  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#hostname"
    required="true"
    index="true" />
  <inputProperty
    propPath="http://capri.csail.mit.edu/2006/capri/common#ipAddrs"
    required="true" />

  <outputBelief
    propPath="http://capri.csail.mit.edu/2006/capri/core#status">
    <fromEvidence
  propPath="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupTest|
            http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult"
    />
  </outputBelief>
  <outputObservation
  propPath="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupTest|
            http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult"
  />
</serviceAdvertisement>

<serviceAdvertisement
  serviceID="knowledge:codns"
  time="1170697054"
  messageType="knowledgeRequest"
  agentURI="http://localhost/codnsAgent"
  cost="2000"
  requesterType="regional"
>

  <outputKnowledge
   subject="http://capri.csail.mit.edu/2006/capri/planetlab#CoDNS_Lookup_Test"
   property="http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult"
  >
     <parent
       propPath="http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
```

```
                http://capri.csail.mit.edu/2006/capri/core#status" />
  </outputKnowledge>

</serviceAdvertisement>
</directoryUpdate>
```

B.3 Dependency knowledge

B.3.1 Manually specified knowledge

At the start of my experiments, the knowledge agent provides the following manually specified knowledge.

```
<knowledge expires='1176096867062'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#HTTP_Server</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer
  </property>
  <cpt>
    <parents>http://capri.csail.mit.edu/2006/capri/core#status</parents>
    <entry p='0.100000' parentVals='FAIL' value='0'/>
    <entry p='0.940000' parentVals='OK'   value='0'/>

    <entry p='0.200000' parentVals='FAIL' value='1'/>
    <entry p='0.050000' parentVals='OK'   value='1'/>

    <entry p='0.300000' parentVals='FAIL' value='2'/>
    <entry p='0.009000' parentVals='OK'   value='2'/>

    <entry p='0.400000' parentVals='FAIL' value='3'/>
    <entry p='0.001000' parentVals='OK'   value='3'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867001'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#HTTP_Server</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.020000' value='FAIL'/>
    <entry p='0.980000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867156'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
  </subject>
```

```
<property>http://capri.csail.mit.edu/2006/capri/core#status</property>
<cpt>
  <parents>
    http://capri.csail.mit.edu/2006/capri/common#localNet|
    http://capri.csail.mit.edu/2006/capri/core#status

    http://capri.csail.mit.edu/2006/capri/common#httpServer|
    http://capri.csail.mit.edu/2006/capri/core#status

    http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
    http://capri.csail.mit.edu/2006/capri/core#status

    http://capri.csail.mit.edu/2006/capri/common#ipRouting|
    http://capri.csail.mit.edu/2006/capri/core#status
  </parents>
  <entry p='1.000000' parentVals='FAIL FAIL FAIL FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL FAIL FAIL OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL FAIL OK   FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL FAIL OK   OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL OK   FAIL FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL OK   FAIL OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL OK   OK   FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='FAIL OK   OK   OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='OK   FAIL FAIL FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='OK   FAIL FAIL OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='OK   FAIL OK   FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='OK   FAIL OK   OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='OK   OK   FAIL FAIL' value='FAIL'/>
  <entry p='1.000000' parentVals='OK   OK   FAIL OK'   value='FAIL'/>
  <entry p='1.000000' parentVals='OK   OK   OK   FAIL' value='FAIL'/>
  <entry p='0.000000' parentVals='OK   OK   OK   OK'   value='FAIL'/>

  <entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='OK'/>
  <entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='OK'/>
  <entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='OK'/>
  <entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='OK'/>
  <entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='OK'/>
  <entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='OK'/>
  <entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='OK'/>
  <entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='OK'/>
  <entry p='0.000000' parentVals='OK   FAIL FAIL FAIL' value='OK'/>
  <entry p='0.000000' parentVals='OK   FAIL FAIL OK'   value='OK'/>
  <entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='OK'/>
  <entry p='0.000000' parentVals='OK   FAIL OK   OK'   value='OK'/>
  <entry p='0.000000' parentVals='OK   OK   FAIL FAIL' value='OK'/>
  <entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='OK'/>
  <entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='OK'/>
```

```
    <entry p='1.000000' parentVals='OK   OK   OK   OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867277'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test
  </subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#httpServer|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#ipRouting|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.010000' parentVals='FAIL FAIL FAIL FAIL' value='0'/>
    <entry p='0.010000' parentVals='FAIL FAIL FAIL OK'   value='0'/>
    <entry p='0.010000' parentVals='FAIL FAIL OK   FAIL' value='0'/>
    <entry p='0.010000' parentVals='FAIL FAIL OK   OK'   value='0'/>
    <entry p='0.010000' parentVals='FAIL OK   FAIL FAIL' value='0'/>
    <entry p='0.010000' parentVals='FAIL OK   FAIL OK'   value='0'/>
    <entry p='0.010000' parentVals='FAIL OK   OK   FAIL' value='0'/>
    <entry p='0.010000' parentVals='FAIL OK   OK   OK'   value='0'/>
    <entry p='0.010000' parentVals='OK   FAIL FAIL FAIL' value='0'/>
    <entry p='0.020000' parentVals='OK   FAIL FAIL OK'   value='0'/>
    <entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='0'/>
    <entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='0'/>
    <entry p='0.010000' parentVals='OK   OK   FAIL FAIL' value='0'/>
    <entry p='0.010000' parentVals='OK   OK   FAIL OK'   value='0'/>
    <entry p='0.010000' parentVals='OK   OK   OK   FAIL' value='0'/>
    <entry p='0.950000' parentVals='OK   OK   OK   OK'   value='0'/>

    <entry p='0.010000' parentVals='FAIL FAIL FAIL FAIL' value='13'/>
    <entry p='0.010000' parentVals='FAIL FAIL FAIL OK'   value='13'/>
```

```
<entry p='0.010000' parentVals='FAIL FAIL OK   FAIL' value='13'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   OK'   value='13'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL FAIL' value='13'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL OK  ' value='13'/>
<entry p='0.010000' parentVals='FAIL OK   OK   FAIL' value='13'/>
<entry p='0.010000' parentVals='FAIL OK   OK   OK  ' value='13'/>
<entry p='0.100000' parentVals='OK   FAIL FAIL FAIL' value='13'/>
<entry p='0.050000' parentVals='OK   FAIL FAIL OK  ' value='13'/>
<entry p='0.100000' parentVals='OK   FAIL OK   FAIL' value='13'/>
<entry p='0.250000' parentVals='OK   FAIL OK   OK  ' value='13'/>
<entry p='0.020000' parentVals='OK   OK   FAIL FAIL' value='13'/>
<entry p='0.020000' parentVals='OK   OK   FAIL OK'   value='13'/>
<entry p='0.150000' parentVals='OK   OK   OK   FAIL' value='13'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK  ' value='13'/>

<entry p='0.010000' parentVals='FAIL FAIL FAIL FAIL' value='14'/>
<entry p='0.010000' parentVals='FAIL FAIL FAIL OK'   value='14'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   FAIL' value='14'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   OK'   value='14'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL FAIL' value='14'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL OK'   value='14'/>
<entry p='0.010000' parentVals='FAIL OK   OK   FAIL' value='14'/>
<entry p='0.010000' parentVals='FAIL OK   OK   OK'   value='14'/>
<entry p='0.050000' parentVals='OK   FAIL FAIL FAIL' value='14'/>
<entry p='0.050000' parentVals='OK   FAIL FAIL OK'   value='14'/>
<entry p='0.250000' parentVals='OK   FAIL OK   FAIL' value='14'/>
<entry p='0.050000' parentVals='OK   FAIL OK   OK'   value='14'/>
<entry p='0.050000' parentVals='OK   OK   FAIL FAIL' value='14'/>
<entry p='0.010000' parentVals='OK   OK   FAIL OK'   value='14'/>
<entry p='0.200000' parentVals='OK   OK   OK   FAIL' value='14'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='14'/>

<entry p='0.010000' parentVals='FAIL FAIL FAIL FAIL' value='20'/>
<entry p='0.010000' parentVals='FAIL FAIL FAIL OK'   value='20'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   FAIL' value='20'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   OK'   value='20'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL FAIL' value='20'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL OK'   value='20'/>
<entry p='0.010000' parentVals='FAIL OK   OK   FAIL' value='20'/>
<entry p='0.010000' parentVals='FAIL OK   OK   OK'   value='20'/>
<entry p='0.050000' parentVals='OK   FAIL FAIL FAIL' value='20'/>
<entry p='0.050000' parentVals='OK   FAIL FAIL OK'   value='20'/>
<entry p='0.150000' parentVals='OK   FAIL OK   FAIL' value='20'/>
<entry p='0.050000' parentVals='OK   FAIL OK   OK'   value='20'/>
<entry p='0.050000' parentVals='OK   OK   FAIL FAIL' value='20'/>
<entry p='0.020000' parentVals='OK   OK   FAIL OK'   value='20'/>
<entry p='0.050000' parentVals='OK   OK   OK   FAIL' value='20'/>
```

```
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='20'/>

<entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='21'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='21'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='21'/>
<entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='21'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL FAIL' value='21'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL OK'   value='21'/>
<entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='21'/>
<entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='21'/>
<entry p='0.000000' parentVals='OK   OK   FAIL FAIL' value='21'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='21'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='21'/>

<entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='22'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='22'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='22'/>
<entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='22'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL FAIL' value='22'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL OK'   value='22'/>
<entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='22'/>
<entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='22'/>
<entry p='0.000000' parentVals='OK   OK   FAIL FAIL' value='22'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='22'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='22'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='22'/>

<entry p='0.100000' parentVals='FAIL FAIL FAIL FAIL' value='2'/>
<entry p='0.100000' parentVals='FAIL FAIL FAIL OK'   value='2'/>
<entry p='0.100000' parentVals='FAIL FAIL OK   FAIL' value='2'/>
<entry p='0.100000' parentVals='FAIL FAIL OK   OK'   value='2'/>
<entry p='0.100000' parentVals='FAIL OK   FAIL FAIL' value='2'/>
<entry p='0.100000' parentVals='FAIL OK   FAIL OK'   value='2'/>
<entry p='0.100000' parentVals='FAIL OK   OK   FAIL' value='2'/>
<entry p='0.100000' parentVals='FAIL OK   OK   OK'   value='2'/>
<entry p='0.390000' parentVals='OK   FAIL FAIL FAIL' value='2'/>
<entry p='0.430000' parentVals='OK   FAIL FAIL OK'   value='2'/>
<entry p='0.420000' parentVals='OK   FAIL OK   FAIL' value='2'/>
```

```
<entry p='0.570000' parentVals='OK   FAIL OK   OK'   value='2'/>
<entry p='0.390000' parentVals='OK   OK   FAIL FAIL' value='2'/>
<entry p='0.390000' parentVals='OK   OK   FAIL OK'   value='2'/>
<entry p='0.440000' parentVals='OK   OK   OK   FAIL' value='2'/>
<entry p='0.040000' parentVals='OK   OK   OK   OK'   value='2'/>

<entry p='0.850000' parentVals='FAIL FAIL FAIL FAIL' value='30'/>
<entry p='0.850000' parentVals='FAIL FAIL FAIL OK'   value='30'/>
<entry p='0.850000' parentVals='FAIL FAIL OK   FAIL' value='30'/>
<entry p='0.850000' parentVals='FAIL FAIL OK   OK'   value='30'/>
<entry p='0.850000' parentVals='FAIL OK   FAIL FAIL' value='30'/>
<entry p='0.850000' parentVals='FAIL OK   FAIL OK'   value='30'/>
<entry p='0.850000' parentVals='FAIL OK   OK   FAIL' value='30'/>
<entry p='0.850000' parentVals='FAIL OK   OK   OK'   value='30'/>
<entry p='0.340000' parentVals='OK   FAIL FAIL FAIL' value='30'/>
<entry p='0.340000' parentVals='OK   FAIL FAIL OK'   value='30'/>
<entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='30'/>
<entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='30'/>
<entry p='0.440000' parentVals='OK   OK   FAIL FAIL' value='30'/>
<entry p='0.510000' parentVals='OK   OK   FAIL OK'   value='30'/>
<entry p='0.010000' parentVals='OK   OK   OK   FAIL' value='30'/>
<entry p='0.010000' parentVals='OK   OK   OK   OK'   value='30'/>

<entry p='0.010000' parentVals='FAIL FAIL FAIL FAIL' value='42'/>
<entry p='0.010000' parentVals='FAIL FAIL FAIL OK'   value='42'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   FAIL' value='42'/>
<entry p='0.010000' parentVals='FAIL FAIL OK   OK'   value='42'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL FAIL' value='42'/>
<entry p='0.010000' parentVals='FAIL OK   FAIL OK'   value='42'/>
<entry p='0.010000' parentVals='FAIL OK   OK   FAIL' value='42'/>
<entry p='0.010000' parentVals='FAIL OK   OK   OK'   value='42'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL FAIL' value='42'/>
<entry p='0.010000' parentVals='OK   FAIL FAIL OK'   value='42'/>
<entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='42'/>
<entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='42'/>
<entry p='0.010000' parentVals='OK   OK   FAIL FAIL' value='42'/>
<entry p='0.010000' parentVals='OK   OK   FAIL OK'   value='42'/>
<entry p='0.010000' parentVals='OK   OK   OK   FAIL' value='42'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='42'/>

<entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='71'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='71'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='71'/>
<entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='71'/>
```

```
    <entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='71'/>
    <entry p='0.010000' parentVals='OK   FAIL FAIL FAIL' value='71'/>
    <entry p='0.010000' parentVals='OK   FAIL FAIL OK'   value='71'/>
    <entry p='0.010000' parentVals='OK   FAIL OK   FAIL' value='71'/>
    <entry p='0.010000' parentVals='OK   FAIL OK   OK'   value='71'/>
    <entry p='0.010000' parentVals='OK   OK   FAIL FAIL' value='71'/>
    <entry p='0.010000' parentVals='OK   OK   FAIL OK'   value='71'/>
    <entry p='0.010000' parentVals='OK   OK   OK   FAIL' value='71'/>
    <entry p='0.000000' parentVals='OK   OK   OK   OK'   value='71'/>

    <entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='72'/>
    <entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='72'/>
    <entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='72'/>
    <entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='72'/>
    <entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='72'/>
    <entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='72'/>
    <entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='72'/>
    <entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='72'/>
    <entry p='0.020000' parentVals='OK   FAIL FAIL FAIL' value='72'/>
    <entry p='0.020000' parentVals='OK   FAIL FAIL OK'   value='72'/>
    <entry p='0.020000' parentVals='OK   FAIL OK   FAIL' value='72'/>
    <entry p='0.020000' parentVals='OK   FAIL OK   OK'   value='72'/>
    <entry p='0.020000' parentVals='OK   OK   FAIL FAIL' value='72'/>
    <entry p='0.020000' parentVals='OK   OK   FAIL OK'   value='72'/>
    <entry p='0.020000' parentVals='OK   OK   OK   FAIL' value='72'/>
    <entry p='0.000000' parentVals='OK   OK   OK   OK'   value='72'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867041'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#Local_Network</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser
  </property>
  <cpt>
    <parents>http://capri.csail.mit.edu/2006/capri/core#status</parents>
    <entry p='0.500000' parentVals='FAIL' value='0'/>
    <entry p='0.950000' parentVals='OK'   value='0'/>

    <entry p='0.300000' parentVals='FAIL' value='1'/>
    <entry p='0.040000' parentVals='OK'   value='1'/>

    <entry p='0.150000' parentVals='FAIL' value='2'/>
    <entry p='0.009000' parentVals='OK'   value='2'/>

    <entry p='0.050000' parentVals='FAIL' value='3'/>
    <entry p='0.001000' parentVals='OK'   value='3'/>
  </cpt>
```

```
</knowledge>
<knowledge expires='1176096866989'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#Local_Network</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.050000' value='FAIL'/>
    <entry p='0.950000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096866995'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.050000' value='FAIL'/>
    <entry p='0.950000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867051'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test
  </subject>
  <property>http://capri.csail.mit.edu/2006/capri/common#probeResult</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='1.000000' parentVals='FAIL' value='FAIL'/>
    <entry p='0.020000' parentVals='OK'   value='FAIL'/>

    <entry p='0.000000' parentVals='FAIL' value='OK'/>
    <entry p='0.980000' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867016'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#IP_Routing</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asPath|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.990000' parentVals='FAIL' value='FAIL'/>
```

```
    <entry p='0.001000' parentVals='OK'   value='FAIL'/>

    <entry p='0.010000' parentVals='FAIL' value='OK'/>
    <entry p='0.999000' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867031'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#IP_Routing</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.080000' value='FAIL'/>
    <entry p='0.920000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867072'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Verify_DNS_Lookup_Test
  </subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.090000' parentVals='FAIL' value='ALIAS'/>
    <entry p='0.550000' parentVals='OK'   value='ALIAS'/>

    <entry p='0.010000' parentVals='FAIL' value='CORRECT'/>
    <entry p='0.418000' parentVals='OK'   value='CORRECT'/>

    <entry p='0.100000' parentVals='FAIL' value='INCORRECT'/>
    <entry p='0.001000' parentVals='OK'   value='INCORRECT'/>

    <entry p='0.700000' parentVals='FAIL' value='LOOKUP_ERROR_CONFIRMED'/>
    <entry p='0.010000' parentVals='OK'   value='LOOKUP_ERROR_CONFIRMED'/>

    <entry p='0.050000' parentVals='FAIL' value='LOOKUP_ERROR_UNCONFIRMED'/>
    <entry p='0.020000' parentVals='OK'   value='LOOKUP_ERROR_UNCONFIRMED'/>

    <entry p='0.050000' parentVals='FAIL' value='LOOKUP_ERROR'/>
    <entry p='0.001000' parentVals='OK'   value='LOOKUP_ERROR'/>
  </cpt>
</knowledge>
```

```
<knowledge expires='1176096867086'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#nextASPath|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#nextASHop|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='1.000000' parentVals='FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   FAIL' value='FAIL'/>
    <entry p='0.000000' parentVals='OK   OK'   value='FAIL'/>

    <entry p='0.000000' parentVals='FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL' value='OK'/>
    <entry p='1.000000' parentVals='OK   OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867112'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.002000' value='FAIL'/>
    <entry p='0.998000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867124'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path_Test</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#asPathTestResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asPath|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.999000' parentVals='FAIL' value='FAIL'/>
    <entry p='0.020000' parentVals='OK'   value='FAIL'/>

    <entry p='0.001000' parentVals='FAIL' value='OK'/>
    <entry p='0.980000' parentVals='OK'   value='OK'/>
```

```
  </cpt>
</knowledge>
<knowledge expires='1176096867138'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Hop</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.100000' value='FAIL'/>
    <entry p='0.900000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1176096867144'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#asHopTestResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asHop|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.995000' parentVals='FAIL' value='FAIL'/>
    <entry p='0.020000' parentVals='OK'   value='FAIL'/>

    <entry p='0.005000' parentVals='FAIL' value='OK'/>
    <entry p='0.980000' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
```

B.3.2 Learned knowledge

The learned probabilistic dependency knowledge produced by the knowledge agent is as follows.

```
<knowledge expires='1175901865431'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#HTTP_Server</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresToServer
  </property>
  <cpt>
    <parents>http://capri.csail.mit.edu/2006/capri/core#status</parents>
    <entry p='0.142081' parentVals='FAIL' value='0'/>
    <entry p='0.725422' parentVals='OK'   value='0'/>

    <entry p='0.298753' parentVals='FAIL' value='1'/>
    <entry p='0.245047' parentVals='OK'   value='1'/>
```

```
    <entry p='0.485112' parentVals='FAIL' value='2'/>
    <entry p='0.026872' parentVals='OK'   value='2'/>

    <entry p='0.074054' parentVals='FAIL' value='3'/>
    <entry p='0.002660' parentVals='OK'   value='3'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865338'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#HTTP_Server</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.263687' value='FAIL'/>
    <entry p='0.736313' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865275'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#HTTP_Connection
  </subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpServer|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#ipRouting|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='1.000000' parentVals='FAIL FAIL FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL FAIL FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL FAIL OK   FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL FAIL OK   OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK   FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK   FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK   OK   FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK   OK   OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   FAIL FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='OK   FAIL FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   FAIL OK   FAIL' value='FAIL'/>
```

```
    <entry p='1.000000' parentVals='OK   FAIL OK   OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   OK   FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='OK   OK   FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   OK   OK   FAIL' value='FAIL'/>
    <entry p='0.000000' parentVals='OK   OK   OK   OK'   value='FAIL'/>

    <entry p='0.000000' parentVals='FAIL FAIL FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL FAIL FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='FAIL FAIL OK   FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK   FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK   OK   FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL OK   OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   OK   FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='OK'/>
    <entry p='1.000000' parentVals='OK   OK   OK   OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865425'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Firefox_Error_Test
  </subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#ffoxErrorCode
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#httpServer|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#httpConn|
      http://capri.csail.mit.edu/2006/capri/common#ipRouting|
```

```
  http://capri.csail.mit.edu/2006/capri/core#status
</parents>
<entry p='0.022017' parentVals='FAIL FAIL FAIL FAIL' value='0'/>
<entry p='0.119692' parentVals='FAIL FAIL FAIL OK'   value='0'/>
<entry p='0.133655' parentVals='FAIL FAIL OK   FAIL' value='0'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='0'/>
<entry p='0.134272' parentVals='FAIL OK   FAIL FAIL' value='0'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='0'/>
<entry p='0.016216' parentVals='FAIL OK   OK   FAIL' value='0'/>
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='0'/>
<entry p='0.064925' parentVals='OK   FAIL FAIL FAIL' value='0'/>
<entry p='0.084086' parentVals='OK   FAIL FAIL OK'   value='0'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='0'/>
<entry p='0.000000' parentVals='OK   FAIL OK   OK'   value='0'/>
<entry p='0.076660' parentVals='OK   OK   FAIL FAIL' value='0'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='0'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='0'/>
<entry p='0.000111' parentVals='OK   OK   OK   OK'   value='0'/>

<entry p='0.060477' parentVals='FAIL FAIL FAIL FAIL' value='13'/>
<entry p='0.136459' parentVals='FAIL FAIL FAIL OK'   value='13'/>
<entry p='0.016756' parentVals='FAIL FAIL OK   FAIL' value='13'/>
<entry p='0.372093' parentVals='FAIL FAIL OK   OK'   value='13'/>
<entry p='0.006871' parentVals='FAIL OK   FAIL FAIL' value='13'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='13'/>
<entry p='0.202455' parentVals='FAIL OK   OK   FAIL' value='13'/>
<entry p='0.182416' parentVals='FAIL OK   OK   OK'   value='13'/>
<entry p='0.082754' parentVals='OK   FAIL FAIL FAIL' value='13'/>
<entry p='0.080527' parentVals='OK   FAIL FAIL OK'   value='13'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='13'/>
<entry p='0.447231' parentVals='OK   FAIL OK   OK'   value='13'/>
<entry p='0.120951' parentVals='OK   OK   FAIL FAIL' value='13'/>
<entry p='0.001003' parentVals='OK   OK   FAIL OK'   value='13'/>
<entry p='0.000904' parentVals='OK   OK   OK   FAIL' value='13'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='13'/>

<entry p='0.168603' parentVals='FAIL FAIL FAIL FAIL' value='14'/>
<entry p='0.101446' parentVals='FAIL FAIL FAIL OK'   value='14'/>
<entry p='0.039133' parentVals='FAIL FAIL OK   FAIL' value='14'/>
<entry p='0.232558' parentVals='FAIL FAIL OK   OK'   value='14'/>
<entry p='0.116590' parentVals='FAIL OK   FAIL FAIL' value='14'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='14'/>
<entry p='0.167762' parentVals='FAIL OK   OK   FAIL' value='14'/>
<entry p='0.137245' parentVals='FAIL OK   OK   OK'   value='14'/>
<entry p='0.150982' parentVals='OK   FAIL FAIL FAIL' value='14'/>
<entry p='0.083361' parentVals='OK   FAIL FAIL OK'   value='14'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='14'/>
```

```
<entry p='0.162278' parentVals='OK   FAIL OK   OK'   value='14'/>
<entry p='0.043511' parentVals='OK   OK   FAIL FAIL' value='14'/>
<entry p='0.003210' parentVals='OK   OK   FAIL OK'   value='14'/>
<entry p='0.997145' parentVals='OK   OK   OK   FAIL' value='14'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='14'/>

<entry p='0.128651' parentVals='FAIL FAIL FAIL FAIL' value='20'/>
<entry p='0.049197' parentVals='FAIL FAIL FAIL OK'   value='20'/>
<entry p='0.058300' parentVals='FAIL FAIL OK   FAIL' value='20'/>
<entry p='0.116279' parentVals='FAIL FAIL OK   OK'   value='20'/>
<entry p='0.144194' parentVals='FAIL OK   FAIL FAIL' value='20'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='20'/>
<entry p='0.026932' parentVals='FAIL OK   OK   FAIL' value='20'/>
<entry p='0.065940' parentVals='FAIL OK   OK   OK'   value='20'/>
<entry p='0.123446' parentVals='OK   FAIL FAIL FAIL' value='20'/>
<entry p='0.067232' parentVals='OK   FAIL FAIL OK'   value='20'/>
<entry p='0.546458' parentVals='OK   FAIL OK   FAIL' value='20'/>
<entry p='0.093284' parentVals='OK   FAIL OK   OK'   value='20'/>
<entry p='0.118216' parentVals='OK   OK   FAIL FAIL' value='20'/>
<entry p='0.001939' parentVals='OK   OK   FAIL OK'   value='20'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='20'/>
<entry p='0.167018' parentVals='OK   OK   OK   OK'   value='20'/>

<entry p='0.080270' parentVals='FAIL FAIL FAIL FAIL' value='21'/>
<entry p='0.097347' parentVals='FAIL FAIL FAIL OK'   value='21'/>
<entry p='0.154658' parentVals='FAIL FAIL OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='21'/>
<entry p='0.056294' parentVals='FAIL OK   FAIL FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='21'/>
<entry p='0.040168' parentVals='FAIL OK   OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='21'/>
<entry p='0.020431' parentVals='OK   FAIL FAIL FAIL' value='21'/>
<entry p='0.142134' parentVals='OK   FAIL FAIL OK'   value='21'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='21'/>
<entry p='0.019329' parentVals='OK   FAIL OK   OK'   value='21'/>
<entry p='0.116575' parentVals='OK   OK   FAIL FAIL' value='21'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='21'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='21'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='21'/>

<entry p='0.132340' parentVals='FAIL FAIL FAIL FAIL' value='22'/>
<entry p='0.057270' parentVals='FAIL FAIL FAIL OK'   value='22'/>
<entry p='0.033964' parentVals='FAIL FAIL OK   FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='22'/>
<entry p='0.026202' parentVals='FAIL OK   FAIL FAIL' value='22'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='22'/>
<entry p='0.084725' parentVals='FAIL OK   OK   FAIL' value='22'/>
```

```
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='22'/>
<entry p='0.095283' parentVals='OK   FAIL FAIL FAIL' value='22'/>
<entry p='0.087302' parentVals='OK   FAIL FAIL OK'   value='22'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='22'/>
<entry p='0.012542' parentVals='OK   FAIL OK   OK'   value='22'/>
<entry p='0.123287' parentVals='OK   OK   FAIL FAIL' value='22'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='22'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='22'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='22'/>

<entry p='0.080474' parentVals='FAIL FAIL FAIL FAIL' value='2'/>
<entry p='0.063688' parentVals='FAIL FAIL FAIL OK'   value='2'/>
<entry p='0.151231' parentVals='FAIL FAIL OK   FAIL' value='2'/>
<entry p='0.232558' parentVals='FAIL FAIL OK   OK'   value='2'/>
<entry p='0.135371' parentVals='FAIL OK   FAIL FAIL' value='2'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='2'/>
<entry p='0.070081' parentVals='FAIL OK   OK   FAIL' value='2'/>
<entry p='0.100208' parentVals='FAIL OK   OK   OK'   value='2'/>
<entry p='0.074100' parentVals='OK   FAIL FAIL FAIL' value='2'/>
<entry p='0.116116' parentVals='OK   FAIL FAIL OK'   value='2'/>
<entry p='0.337416' parentVals='OK   FAIL OK   FAIL' value='2'/>
<entry p='0.257886' parentVals='OK   FAIL OK   OK'   value='2'/>
<entry p='0.022831' parentVals='OK   OK   FAIL FAIL' value='2'/>
<entry p='0.016316' parentVals='OK   OK   FAIL OK'   value='2'/>
<entry p='0.001951' parentVals='OK   OK   OK   FAIL' value='2'/>
<entry p='0.826663' parentVals='OK   OK   OK   OK'   value='2'/>

<entry p='0.077131' parentVals='FAIL FAIL FAIL FAIL' value='30'/>
<entry p='0.101977' parentVals='FAIL FAIL FAIL OK'   value='30'/>
<entry p='0.047207' parentVals='FAIL FAIL OK   FAIL' value='30'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='30'/>
<entry p='0.193109' parentVals='FAIL OK   FAIL FAIL' value='30'/>
<entry p='1.000000' parentVals='FAIL OK   FAIL OK'   value='30'/>
<entry p='0.043161' parentVals='FAIL OK   OK   FAIL' value='30'/>
<entry p='0.482001' parentVals='FAIL OK   OK   OK'   value='30'/>
<entry p='0.071377' parentVals='OK   FAIL FAIL FAIL' value='30'/>
<entry p='0.122080' parentVals='OK   FAIL FAIL OK'   value='30'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='30'/>
<entry p='0.000000' parentVals='OK   FAIL OK   OK'   value='30'/>
<entry p='0.138651' parentVals='OK   OK   FAIL FAIL' value='30'/>
<entry p='0.975460' parentVals='OK   OK   FAIL OK'   value='30'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='30'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='30'/>

<entry p='0.045796' parentVals='FAIL FAIL FAIL FAIL' value='42'/>
<entry p='0.079017' parentVals='FAIL FAIL FAIL OK'   value='42'/>
<entry p='0.140285' parentVals='FAIL FAIL OK   FAIL' value='42'/>
```

```
<entry p='0.046512' parentVals='FAIL FAIL OK   OK'   value='42'/>
<entry p='0.144814' parentVals='FAIL OK   FAIL FAIL' value='42'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='42'/>
<entry p='0.027302' parentVals='FAIL OK   OK   FAIL' value='42'/>
<entry p='0.006577' parentVals='FAIL OK   OK   OK'   value='42'/>
<entry p='0.143356' parentVals='OK   FAIL FAIL FAIL' value='42'/>
<entry p='0.048148' parentVals='OK   FAIL FAIL OK'   value='42'/>
<entry p='0.000004' parentVals='OK   FAIL OK   FAIL' value='42'/>
<entry p='0.000000' parentVals='OK   FAIL OK   OK'   value='42'/>
<entry p='0.109213' parentVals='OK   OK   FAIL FAIL' value='42'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='42'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='42'/>
<entry p='0.000779' parentVals='OK   OK   OK   OK'   value='42'/>

<entry p='0.155340' parentVals='FAIL FAIL FAIL FAIL' value='71'/>
<entry p='0.075350' parentVals='FAIL FAIL FAIL OK'   value='71'/>
<entry p='0.157546' parentVals='FAIL FAIL OK   FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='71'/>
<entry p='0.003059' parentVals='FAIL OK   FAIL FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='71'/>
<entry p='0.195512' parentVals='FAIL OK   OK   FAIL' value='71'/>
<entry p='0.000000' parentVals='FAIL OK   OK   OK'   value='71'/>
<entry p='0.009326' parentVals='OK   FAIL FAIL FAIL' value='71'/>
<entry p='0.126800' parentVals='OK   FAIL FAIL OK'   value='71'/>
<entry p='0.116121' parentVals='OK   FAIL OK   FAIL' value='71'/>
<entry p='0.003670' parentVals='OK   FAIL OK   OK'   value='71'/>
<entry p='0.023946' parentVals='OK   OK   FAIL FAIL' value='71'/>
<entry p='0.000000' parentVals='OK   OK   FAIL OK'   value='71'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='71'/>
<entry p='0.005429' parentVals='OK   OK   OK   OK'   value='71'/>

<entry p='0.048899' parentVals='FAIL FAIL FAIL FAIL' value='72'/>
<entry p='0.118557' parentVals='FAIL FAIL FAIL OK'   value='72'/>
<entry p='0.067265' parentVals='FAIL FAIL OK   FAIL' value='72'/>
<entry p='0.000000' parentVals='FAIL FAIL OK   OK'   value='72'/>
<entry p='0.039225' parentVals='FAIL OK   FAIL FAIL' value='72'/>
<entry p='0.000000' parentVals='FAIL OK   FAIL OK'   value='72'/>
<entry p='0.125685' parentVals='FAIL OK   OK   FAIL' value='72'/>
<entry p='0.025614' parentVals='FAIL OK   OK   OK'   value='72'/>
<entry p='0.164019' parentVals='OK   FAIL FAIL FAIL' value='72'/>
<entry p='0.042215' parentVals='OK   FAIL FAIL OK'   value='72'/>
<entry p='0.000000' parentVals='OK   FAIL OK   FAIL' value='72'/>
<entry p='0.003780' parentVals='OK   FAIL OK   OK'   value='72'/>
<entry p='0.106158' parentVals='OK   OK   FAIL FAIL' value='72'/>
<entry p='0.002073' parentVals='OK   OK   FAIL OK'   value='72'/>
<entry p='0.000000' parentVals='OK   OK   OK   FAIL' value='72'/>
<entry p='0.000000' parentVals='OK   OK   OK   OK'   value='72'/>
```

```
  </cpt>
</knowledge>
<knowledge expires='1175901865438'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#Local_Network</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#consecFailuresFromUser
  </property>
  <cpt>
    <parents>http://capri.csail.mit.edu/2006/capri/core#status</parents>
    <entry p='0.006989' parentVals='FAIL' value='0'/>
    <entry p='0.985049' parentVals='OK'   value='0'/>

    <entry p='0.099216' parentVals='FAIL' value='1'/>
    <entry p='0.003635' parentVals='OK'   value='1'/>

    <entry p='0.608251' parentVals='FAIL' value='2'/>
    <entry p='0.010778' parentVals='OK'   value='2'/>

    <entry p='0.285544' parentVals='FAIL' value='3'/>
    <entry p='0.000538' parentVals='OK'   value='3'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865334'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#Local_Network</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.130702' value='FAIL'/>
    <entry p='0.869298' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865341'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#DNS_Lookup</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.333942' value='FAIL'/>
    <entry p='0.666058' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865444'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Outbound_Conn_Test
  </subject>
  <property>http://capri.csail.mit.edu/2006/capri/common#probeResult</property>
```

```
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#localNet|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.997443' parentVals='FAIL' value='FAIL'/>
    <entry p='0.013569' parentVals='OK'   value='FAIL'/>

    <entry p='0.002557' parentVals='FAIL' value='OK'/>
    <entry p='0.986431' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865353'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#IP_Routing</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asPath|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.996292' parentVals='FAIL' value='FAIL'/>
    <entry p='0.024386' parentVals='OK'   value='FAIL'/>

    <entry p='0.003708' parentVals='FAIL' value='OK'/>
    <entry p='0.975614' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865218'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#IP_Routing</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.010000' value='FAIL'/>
    <entry p='0.990000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865452'>
  <subject>
    http://capri.csail.mit.edu/2006/capri/common#Verify_DNS_Lookup_Test
  </subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#dnsLookupResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
```

```
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.166841' parentVals='FAIL' value='ALIAS'/>
    <entry p='0.248752' parentVals='OK'   value='ALIAS'/>

    <entry p='0.144186' parentVals='FAIL' value='CORRECT'/>
    <entry p='0.596506' parentVals='OK'   value='CORRECT'/>

    <entry p='0.229032' parentVals='FAIL' value='INCORRECT'/>
    <entry p='0.009483' parentVals='OK'   value='INCORRECT'/>

    <entry p='0.310981' parentVals='FAIL' value='LOOKUP_ERROR_CONFIRMED'/>
    <entry p='0.001517' parentVals='OK'   value='LOOKUP_ERROR_CONFIRMED'/>

    <entry p='0.079139' parentVals='FAIL' value='LOOKUP_ERROR_UNCONFIRMED'/>
    <entry p='0.044230' parentVals='OK'   value='LOOKUP_ERROR_UNCONFIRMED'/>

    <entry p='0.069821' parentVals='FAIL' value='LOOKUP_ERROR'/>
    <entry p='0.099512' parentVals='OK'   value='LOOKUP_ERROR'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865247'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#nextASPath|
      http://capri.csail.mit.edu/2006/capri/core#status

      http://capri.csail.mit.edu/2006/capri/common#nextASHop|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='1.000000' parentVals='FAIL FAIL' value='FAIL'/>
    <entry p='1.000000' parentVals='FAIL OK'   value='FAIL'/>
    <entry p='1.000000' parentVals='OK   FAIL' value='FAIL'/>
    <entry p='0.000000' parentVals='OK   OK'   value='FAIL'/>

    <entry p='0.000000' parentVals='FAIL FAIL' value='OK'/>
    <entry p='0.000000' parentVals='FAIL OK'   value='OK'/>
    <entry p='0.000000' parentVals='OK   FAIL' value='OK'/>
    <entry p='1.000000' parentVals='OK   OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865347'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
```

```
    <parents>
    </parents>
    <entry p='0.054708' value='FAIL'/>
    <entry p='0.945292' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865458'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Path_Test</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#asPathTestResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asPath|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.977004' parentVals='FAIL' value='FAIL'/>
    <entry p='0.251472' parentVals='OK'   value='FAIL'/>

    <entry p='0.022996' parentVals='FAIL' value='OK'/>
    <entry p='0.748528' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865267'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Hop</subject>
  <property>http://capri.csail.mit.edu/2006/capri/core#status</property>
  <cpt>
    <parents>
    </parents>
    <entry p='0.001000' value='FAIL'/>
    <entry p='0.999000' value='OK'/>
  </cpt>
</knowledge>
<knowledge expires='1175901865270'>
  <subject>http://capri.csail.mit.edu/2006/capri/common#AS_Hop_Test</subject>
  <property>
    http://capri.csail.mit.edu/2006/capri/common#asHopTestResult
  </property>
  <cpt>
    <parents>
      http://capri.csail.mit.edu/2006/capri/common#asHop|
      http://capri.csail.mit.edu/2006/capri/core#status
    </parents>
    <entry p='0.995000' parentVals='FAIL' value='FAIL'/>
    <entry p='0.020000' parentVals='OK'   value='FAIL'/>

    <entry p='0.005000' parentVals='FAIL' value='OK'/>
```

```
    <entry p='0.980000' parentVals='OK'   value='OK'/>
  </cpt>
</knowledge>
```

B.3.3 CoDNS knowledge

The new CoDNS agent provides the following knowledge. The conditional probabilities are derived from the dependency knowledge for regular DNS lookup tests. These probabilities assume that result of a CoDNS lookup test for a hostname for which the DNS lookup test returns `ALIAS` is `INCORRECT`.

```
<knowledge>
<subject>
  http://capri.csail.mit.edu/2006/capri/planetlab#CoDNS_Lookup_Test
</subject>
<property>
  http://capri.csail.mit.edu/2006/capri/planetlab#codnsLookupResult
</property>
<cpt>
  <parents>
    http://capri.csail.mit.edu/2006/capri/common#dnsLookup|
      http://capri.csail.mit.edu/2006/capri/core#status
  </parents>
  <entry value="CORRECT"                  parentVals="FAIL" p="0.144186" />
  <entry value="INCORRECT"                parentVals="FAIL" p="0.395873" />
  <entry value="LOOKUP_ERROR"             parentVals="FAIL" p="0.069821" />
  <entry value="LOOKUP_ERROR_CONFIRMED"   parentVals="FAIL" p="0.310981" />
  <entry value="LOOKUP_ERROR_UNCONFIRMED" parentVals="FAIL" p="0.079139" />

  <entry value="CORRECT"                  parentVals="OK"   p="0.596506" />
  <entry value="INCORRECT"                parentVals="OK"   p="0.258235" />
  <entry value="LOOKUP_ERROR"             parentVals="OK"   p="0.099512" />
  <entry value="LOOKUP_ERROR_CONFIRMED"   parentVals="OK"   p="0.001517" />
  <entry value="LOOKUP_ERROR_UNCONFIRMED" parentVals="OK"   p="0.044230" />
</cpt>
</knowledge>
```

Bibliography

[1] P. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. A. Maltz, R. Mortier, M. Wawrzoniak, and M. Zhang. Discovering dependencies for network management. In *Proceedings of the Fifth Workshop on Hot Topics in Networks (HotNets-V)*, 2006.

[2] A. Bouloutas, S. Calo, A. Finkel, and I. Katzela. Distributed fault identification in telecommunication networks. *Journal of Network and Systems Management*, 3(3):295–312, 1995.

[3] M. Brodie, I. Rish, and S. Ma. Intelligent probing: A cost-effective approach to fault diagnosis in computer networks. *IBM Systems Journal*, September 2002.

[4] T. Bu, N. Duffield, F. Presti, and D. Towsley. Network tomography on general topologies. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS '02)*, 2002.

[5] K. L. Calvert and J. Griffioen. On information hiding and network management. In *Proceedings of SIGCOMM'06 Workshop on Internet Network Management (INM'06)*, 2006.

[6] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[7] R. C. Chalmers and K. C. Almeroth. Modeling the branching characteristics and efficiency gains of global multicast trees. In *Proceedings of INFOCOM*, pages 449–458, 2001.

[8] B. Chandrasekaran and J. R. Josephson. The ontology of tasks and methods. In *Proceedings of AAAI 1997 Spring Symposium on Ontological Engineering*, 1997.

[9] B. Chandrasekaran and S. Mittal. On deep versus compiled knowledge approaches to medical diagnosis. In *Proceedings of American Association for Artificial Intelligence Conference*, pages 349–354, August 1982.

[10] C. S. Chao, D. L. Yang, and A. C. Liu. An automated fault diagnosis system using hierarchical reasoning and alarm correlation. *Journal of Network and Systems Management*, 9(2):183–202, June 2001.

[11] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings of International Conference on Dependable Systems and Networks*, pages 595–604, 2002.

[12] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the Internet. In *Proceedings of SIGCOMM '03*, 2003.

[13] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, pages 231–244. USENIX Association, 2004.

[14] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 105–118, New York, NY, USA, 2005. ACM Press.

[15] J. M. Crawford, D. L. Dvorak, D. J. Litman, A. K. Mishra, and P. F. Patel-Schneider. Device representation and reasoning with affective relations. In *Proceedings of IJCAI '95*, 1995.

[16] J. M. Crawford and B. J. Kuipers. Algernon — a tractable system for knowledge representation. *SIGART Bulletin*, 2(3):35–44, June 1991.

[17] C. Crick and A. Pfeffer. Loopy belief propagation as a basis for communication in sensor networks. In *Proceedings of the 19th Annual Conference on Uncertainty in Artificial Intelligence (UAI-03)*, 2003.

[18] A. Darwiche. Model-based diagnosis using structured system descriptions. *Journal of Artificial Intelligence Research*, 8:165–222, 1998.

[19] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24(1-3):347–410, 1984.

[20] J. de Kleer and O. Raiman. Trading off the costs of inference vs. probing in diagnosis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 2, pages 1736–1741, Montreal, Quebec, Canada, August 1995. Morgan Kaufmann.

[21] J. de Kleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:100–117, 1987.

[22] J. de Kleer and B. Williams. Diagnosis with behavioral modes. In *Proceedings of the 11th International Joint Conference on AI (IJCAI-89)*, Detroit, MI, USA, 1989.

[23] M. Dean, G. Schreiber, S. Bechhofer, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. OWL Web Ontology Language reference. W3C Recommendation, February 2004.

[24] R. H. Deng, A. A. Lazar, and W. Wang. A probabilistic approach to fault diagnosis in linear lightwave networks. *IEEE Journal of Selected Areas in Communications*, 11(9):1438–1448, 1993.

[25] S. L. Dittmer and F. V. Jensen. Myopic value of information in influence diagrams. In *Proc. of the 13th Conference on UAI*, pages 142–149, 1997.

[26] A. Doucet, N. de Freitas, K. P. Murphy, and S. J. Russell. Rao-Blackwellised particle filtering for dynamic Bayesian networks. In *UAI '00: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, pages 176–183, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[27] E. Ekaette and B. Far. A framework for distributed network fault management using intelligent network agents. In *Proceedings of IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, pages 797–800, May 2003.

[28] P. T. H. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 2003.

[29] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[30] D. Fensel and R. Benjamins. Assumptions in model-based diagnosis. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-based Systems workshop (KAW '96)*, 1996.

[31] M. P. Féret and J. I. Glasgow. Combining case-based and model-based reasoning for the diagnosis of complex devices. *Applied Intelligence*, 7:57–78, 1997.

[32] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A pervasive network tracing framework. In *Proceedings of the Fourth USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*, Boston, MA, USA, 2007.

[33] N. Friedman, L. Getoor, D. Koller, and A. Pfeffer. Learning probabilistic relational models. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, Stockholm, Sweden, August 1999.

[34] R. Gopal. Layered model for supporting fault isolation and recovery. In *Proceedings of NOMS*, pages 729–742, 2000.

[35] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5), October 2005.

[36] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of DSOM '98*, 1998.

[37] A. K. M. Guangtian Liu and E. Yang. Composite events for network event correlation. In *Proc. of IFIP/IEEE International Symposium on Integrated Network Management*, May 1999.

[38] M. Gupta, A. Neogi, M. K. Agarwal, and G. Kar. *Self-Managing Distributed Systems*, volume 2867/2004 of *Lecture Notes in Computer Science*, chapter Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination, pages 221–233. Springer Berlin / Heidelberg, 2003.

[39] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. IETF RFC 2608, June 1999.

[40] M. Hasan and B. Sugla. A conceptual framework for network management event correlation and filtering systems. In *Proceedings of IEEE/IFIP International Conference on Integrated Network Management*, pages 233–246, May 1999.

[41] C. S. Hood and C. Ji. Proactive network fault detection. In *Proceedings of INFOCOM (3)*, pages 1147–1155, 1997.

[42] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00)*, Boston, MA, USA, August 2000.

[43] F. Jensen and J. Liang. drHugin A system for value of information in Bayesian networks. In *Proceedings of the Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems*, 1994.

[44] S. Kandula, D. Katabi, and J.-P. Vasseur. Shrink: A tool for failure diagnosis in IP networks. In *ACM SIGCOMM Workshop on mining network data (MineNet-05)*, Philadelphia, PA, USA, August 2005.

[45] Katzela and Schwartz. Schemes for fault identification in communication networks. *IEEE/ACM Transactions on Networking*, 3, 1995.

[46] A. Keller, U. Blumenthal, and G. Kar. Classification and computation of dependencies for distributed management. In *Proceedings of the Fifth International Conference on Computers and Communications*, 2000.

[47] E. Kiciman and L. Subramanian. A root cause localization model for large scale systems. In *Proceedings of Workshop on Hot Topics in Systems Dependability (HotDep)*, 2005.

[48] Y.-G. Kim and M. Valtorta. On the detection of conflicts in diagnostic Bayesian networks using abstraction. In *Proceedings of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 362–367, San Francisco, CA, USA, 1995. Morgan Kaufmann.

[49] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. In *Proceedings of the fourth international symposium on Integrated network management IV*, pages 266–277, London, UK, 1995. Chapman & Hall, Ltd.

[50] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP fault localization via risk modeling. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[51] A. Krause and C. Guestrin. Optimal nonmyopic value of information in graphical models — efficient algorithms and theoretical limits. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 1339–1345, August 2005.

[52] B. Krishnamachari, D. Estrin, and S. Wicker. The impact of data aggregation in wireless sensor networks. In *Proceedings of ICDCS Workshops*, 2002.

[53] J. Kurien, X. Koutsoukos, and F. Zhao. Distributed diagnosis of networked, embedded systems. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002)*, pages 179–188, 2002.

[54] S. Ktker. A modeling framework for integrated distributed systems fault management. In C. Popien, editor, *Proceedings of IFIP/IEEE International Conference on Distributed Platforms*, pages 187–198, Dresden, Germany, 1996.

[55] J. Lauber, C. Steger, and R. Weiss. Autonomous agents for online diagnosis of a safety-critical system based on probabilistic causal reasoning. In *Proceedings of the Fourth International Symposium on Autonomous Decentralized Systems*, page 213, 1999.

[56] G. Lee, P. Faratin, S. Bauer, and J. Wroclawski. A user-guided cognitive agent for network service selection in pervasive computing environments. In *Proceedings of Second IEEE International Conference on Pervasive Computing and Communications (PerCom '04)*, 2004.

[57] G. J. Lee and L. Poole. Diagnosis of TCP overlay connection failures using Bayesian networks. In *Proceedings of the SIGCOMM 2006 Workshop on Mining Network Data (MineNet-06)*, September 2006.

[58] P. P. C. Lee, V. Misra, and D. Rubenstein. Toward optimal network fault correction via end-to-end inference. Technical report, Columbia University, May 2006.

[59] U. Lerner, R. Parr, D. Koller, and G. Biswas. Bayesian fault detection and diagnosis in dynamic systems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 531–537, Austin, TX, USA, August 2000.

[60] L. M. Lewis. A case-based reasoning approach to the resolution of faults in communication networks. In *Proceedings of the IFIP TC6/WG6.6 Third International Symposium on Integrated Network Management*, pages 671–682. North-Holland, 1993.

[61] H. Li and J. S. Baras. A framework for supporting intelligent fault and performance management for communication networks. In *Proceedings of 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services (MNS '01),*, 2001.

[62] M. L. Littman, N. Ravi, E. Fenson, and R. Howard. Reinforcement learning for autonomic network repair. In *Proceedings of the First International Conference on Autonomic Computing (ICAC '04)*, 2004.

[63] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proceedings of the Seventh USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)*, November 2006.

[64] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet path diagnosis. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[65] T. P. Minka. Expectation propagation for approximate Bayesian inference. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 362–369, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[66] B. P. N. Dawes, J. Altoft. Network diagnosis by reasoning in uncertain nested evidence spaces. *IEEE Transactions on Communications*, 43(2/3/4):466–476, 1995.

[67] V. N. Padmanabhan, S. Ramabhadran, and J. Padhye. NetProfiler: Profiling wide-area networks using peer cooperation. In *Proceedings of the Fourth International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2005.

[68] V. Pappas, P. Fältström, D. Massey, and L. Zhang. Distributed DNS troubleshooting. In *NetT '04: Proceedings of the ACM SIGCOMM workshop on Network troubleshooting*, pages 265–270, New York, NY, USA, 2004. ACM Press.

[69] K. Park, V. S. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.

[70] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[71] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[72] J. Pinkston, J. Undercoffer, A. Joshi, and T. Finin. A target-centric ontology for intrusion detection. *Knowledge Engineering Review*, 2004.

[73] S. Quirolgico, P. Assis, A. Westerinen, M. Baskey, and E. Stokes. Toward a formal common information model ontology. In *WISE 2004 Workshops*, pages 11–21, 2004.

[74] L. D. Raedt and K. Kersting. Probabilistic logic learning. *ACM-SIGKDD Explorations*, 2003.

[75] D. T. Rami Debouk, Stéphane Lafortune. Coordinated decentralized protocols for failure diagnosis of discrete event systems. *Journal of Discrete Event Dynamical Systems*, 2000.

[76] I. Rish, M. Brodie, N. Odintsova, S. Ma, and G. Grabarnik. Real-time problem determination in distributed systems using active probing. In *Proceedings of Network Operations and Management Symposium (NOMS 2004)*, 2004.

[77] T. Roscoe, R. Mortier, P. Jardetzky, and S. Hand. InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems. In *Proceedings of the 2002 ACM SIGOPS European Workshop*, Saint-Emilion, France, September 2002.

[78] S. Sanghai, P. Domingos, and D. Weld. Relational dynamic Bayesian networks. *Journal of Artificial Intelligence Research*, 24:1–39, September 2005.

[79] N. Shadbolt, T. Berners-Lee, and W. Hall. The Semantic Web revisited. *IEEE Intelligent Systems*, 21(3):96–101, May/June 2006.

[80] P. Smyth. Markov monitoring with unknown states. *IEEE Journal on Selected Areas in Communications, special issue on intelligent signal processing for communications*, 12(9), December 1994.

[81] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A public Internet measurement facility. In *Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[82] M. Steinder and A. Sethi. Increasing robustness of fault localization through analysis of lost, spurious, and positive symptoms. In *Proceedings of INFOCOM*, 2002.

[83] M. Steinder and A. S. Sethi. End-to-end service failure diagnosis using belief networks. In *Proceedings of Network Operation and Management Symposium*, 2002.

[84] M. Steinder and A. S. Sethi. Multi-domain diagnosis of end-to-end service failures in hierarchically routed networks. In *Proceedings of Networking 2004*, 2004.

[85] M. Steinder and A. S. Sethi. Probabilistic fault diagnosis in communication systems through incremental hypothesis updating. *Computer Networks*, 45(4):537–562, 2004.

[86] M. Steinder and A. S. Sethi. Probabilistic fault localization in communication systems using belief networks. *IEEE/ACM Transactions on Networking*, 12(5):809–822, 2004.

[87] M. Steinder and A. S. Sethi. A survey of fault localization techniques in computer networks. *Science of Computer Programming*, 53(2):165–194, November 2004.

[88] M. Steinder and A. S. Sethi. Multidomain diagnosis of end-to-end service failures in hierarchically routed networks. *IEEE Transactions on Parallel and Distributed Systems*, 18(3):379–392, 2007.

[89] D. G. Thaler and C. V. Ravishankar. An architecture for inter-domain troubleshooting. *Journal of Network and Systems Management*, 12(2), 2004.

[90] Y. Tsang, M. Coates, and R. Nowak. Passive unicast network tomography using EM algorithms. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, Utah, May 2001.

[91] C. Wang and M. Schwartz. Identification of faulty links in dynamic-routed networks. *IEEE Journal on Selected Areas in Communications*, 11(9):1449–1460, Dec 1993.

[92] A. Ward, P. Glynn, and K. Richardson. Internet service performance failure detection. *SIGMETRICS Perform. Eval. Rev.*, 26(3):38–43, 1998.

[93] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: an information plane for networked systems. *ACM SIGCOMM Computer Communication Review*, 34(1):15–20, 2004.

[94] C. Williamson. On filter effects in web caching hierarchies. *ACM Transactions on Internet Technology*, 2(1), 2002.

[95] P. Wu, R. Bhatnagar, L. Epshtein, M. Bhandaru, and Z. Shi. Alarm correlation engine (ACE). In *Proceedings of Network Operations and Management Symposium (NOMS '98)*, volume 3, pages 733–742, Feb 1998.

[96] M. Yajnik, J. Kurose, and D. Towsley. Packet loss correlation in the MBone multicast network experimental measurements and Markov chain models. Technical Report UM-CS-1995-115, University of Massachusetts, 1995.

[97] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996.

[98] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *Proceedings of Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, 2004.