InfoSpect: Using a Logic Language for System Health Monitoring in Distributed Systems

Timothy Roscoe
Intel Research at Berkeley
2150 Shattuck Avenue, Suite 1300
Berkeley, CA, 94704, USA
troscoe@intel-research.net

Richard Mortier
Microsoft Research
7 J.J. Thompson Avenue
Cambridge CB3 0FB, UK
mort@microsoft.com

Paul Jardetzky
Sprint Labs
1 Adrian Court
Burlingame, CA 94010, USA
pjardetzky@sprintlabs.com

Steven Hand
Univ. Cambridge Computer Laboratory
15 J.J. Thompson Avenue
Cambridge CB3 0FD, UK
steven.hand@cl.cam.ac.uk

Abstract

Dependable systems cannot be built without a monitoring and management component. In this paper we propose using a wide variety of information gathering tools coupled with custom scripts and a Prolog language engine to aggregate information from multiple sources. Complex queries, difficult to express in standard database languages, can then be used to answer questions about the system (e.g. the health of individual components) or to discover contradictions (e.g. inconsistent configurations). We describe our prototype implementation and present some early results.

1. Introduction

Today's distributed systems are highly complex dynamic environments where components are strongly dependent on each other for their correct behavior. This situation becomes further complicated as we move toward pervasive and mobile computing since systems must now cope with a huge variety of devices and device software versions. In such systems change is no longer rare: indeed, the state of individual components may be changing rapidly enough that inconsistency and instability is the norm. This may be an unavoidable emergent property of sufficiently large and complex distributed systems (see e.g. recent work on Internet routing stability [7, 9, 14]).

Yet it is crucial that these dynamic and intricate environments be managed: service providers need to sell a dependable service to customers; administrators need to provide a solid infrastructure to users; and users need their ubiquitous networks to be unobtrusively reliable. System management—always a difficult problem—becomes particularly acute when the manager has so little control over so many aspects of the environment, or little idea about how it all fits together [2].

This paper addresses the problem of dependability of complex, dynamic distributed systems. We specifically look at the problem of system health monitoring, answering questions like "Is the system functioning correctly?", "What is wrong with the system?", "Is the state of the system in line or at odds with our expectations?", and "What needs to be fixed to ensure correct functioning of the system?"

We start from an assumption that the system will always admit inconsistency (at least transiently), and from the belief that the goal of dependability can be furthered by capturing and reasoning about these inconsistencies. We use a logic programming language to reason about the state of the system, at least in part since the flexible data representation afforded by such languages is rather more appropriate for dealing with inconsistencies than a rigid database schema.

System information is gathered by using whatever ad hoc or off-the-shelf tools are available; our approach is to build upon existing system and network management tools, not to replace them. Indeed, a key benefit of our approach is the ability to use tools with overlapping areas of applicability, explicitly record the origin of each piece of information gathered, and then reproduce this when resolving contradictions.

In addressing the above problems in this paper, we are explicitly not attempting to provide a mechanism for actively controlling or configuring networks. Neither is InfoSpect intended to provide the kind of rapid response offered by, for example, networking intrusion detection systems. Instead, we focus on providing useful diagnostic information to a management system (which still involves a human being), integrated from a variety of sources, and which can effectively deal with unforeseen and/or contradictory conditions in the system being monitored.

2. Current approaches

Current approaches to determining system state may be split into network-related and host-related schemes. The former tend to be marketed by networking equipment or big-iron vendors and address the traditional FCAPS objectives of network management. Examples include Cisco's CiscoWorks2000, Micromuse's Netcool, Riversoft's OpenRiver, IBM's Tivoli Netview, and the HP OpenView suite.

These systems help operators control their network by providing simplified interfaces to topology discovery, service provisioning and equipment maintenance checks. The Internet community also provides some network-related inspection tools, particularly for checking consistency or syntax of router configuration, for example RPSL at RIPE [13], or the ISI RaToolSet [6].

Commercial host-related systems management software typically focuses on the PC/server oriented enterprise space: e.g. IBM's Tivoli Suite, or Computer Associates' UniCenter. Microsoft's SMS is targetted at smaller systems composed of Windows-based machines.

All these commercial systems, while useful, are insufficient to solve the problems of managing a dependable computing environment since they assume prior knowledge of the correct state of a semi-static set of actors. The diverse, dynamic and ad hoc systems of the future are not well served by such simplifying assumptions.

In the research community, there are efforts to increase system dependability by building operating systems and architectures for distributed and/or ubiquitous computing (e.g. EROS [15], Xenoservers [12], one.world [4], JX [3]). This work is valuable, but we see it as largely orthogonal to our work: no matter how reliable or flexible individual components are, there will always be a need for a distributed monitoring and management function.

To summarize: reliability management based on assumptions of total control within well-defined perimeters starts to look very fragile in the context of dynamically evolving networks of devices and peer-to-peer software systems. To cope with such an environment, management software must itself be ad-hoc and constantly evolving. The remainder of our paper presents our design rationale in further detail, and introduces our prototype implementation and initial results.

3. Our approach: Logic languages

We have codified our motivation for investigating logic languages into a series of (overlapping) design principles which we present below:

Decouple health monitoring from system operation

Most system management tools manage 'before the fact': they tightly integrate the functions of monitoring and control. The emphasis is on deciding on a desired system state, and then making the system elements consistent with that state. While this approach can work well in a highly centralized and controlled environment, complete consistency is an unrealistic goal in a large and complex system. There are several reasons for this:

1. The time taken for the system to converge to the desired state may be comparable with the interval between configuration changes. This has been observed to be the case with many networks and peer-to-peer systems [8, 17].

2. Changes are frequently made independently of the management entity. In many systems, a central management solution simply does not scale socially, since the demands of users for changes to the infrastructure exceed the capacity of the management organization to implement them in a timely manner. As a result, users (whether individuals or organizations) take matters into their own hands. This is not an unusual state of affairs in networking research laboratories, for example. More generally, this is the normal state of affairs in a pervasive computing system.

3. There may be no clearly defined notion of a central management entity anyway, because the system is in constant interaction with others whose configurations are themselves changing. This is again the case for mobile users in a future pervasive computing environment, but is also true for large ISP networks which have peering relationships with other carriers.

An alternative approach to system health monitoring is to manage 'after the fact': construct a view of what the configuration of the system actually *is*, and then allow the operator to manage the system in order to achieve what is desired. This option is more appropriate in the kinds of open and dynamic environments we are interested in here, and our approach falls squarely into this category.

Logic languages such as Prolog [1] seem to have many advantages over the relational databases used in modern system monitoring packages. Prolog is a purely declarative language in which a program consists of *facts* about objects in a system, *inference rules* which define relationships between objects and allow the derivation of new facts, and *queries* which ask questions about objects and their relationships. Facts in Prolog are free-form logical propositions; there is no predefined data schema.

Make it easy to integrate diverse information sources

As well as integrated system management environments, a wide variety of excellent ad hoc system monitoring tools are available both commercially and non-commercially. Such tools work well because they focus on specific functionality – port scanning, SNMP traps, etc. Effective distributed system diagnosis, on the other hand, requires correlating and aggregating information from many sources.

We want to use as many of these sources of information as possible: as systems evolve over time, a system health monitor which can usefully combine the results from other tools will win over an integrated solution which tries to do everything itself.

There are two challenges here: providing a common *representation* of the results from disparate tools so that they can be unified, and the software engineering problem of *interfacing* our monitoring system to these tools.

Prolog performs well in both roles here: in contrast to monolithic software architectures, logic languages are highly effective at unifying the output of other tools via the use of inference rules, and in contrast to RDBM systems, the absence of predefined schema makes it easy to add new information sources.

Expect the unexpected

In large distributed systems, contradictions frequently exist between what the managers believe the system configuration to be and what is independently observed to be the actual system state. These contradictions are usually symptomatic of security problems, faults, or misconfigurations, yet they are unlikely to be detected by monolithic systems based on well defined schema, since the idea of a schema itself always presupposes some consistency of state.

For example, a database which models domain name records as a series of Unix `hostent`-like structures will have no way of representing a situation in which two replica domain name servers have conflicting A-records.

Put simply: relational database systems do not handle contradictions well. The separation between observed system facts and inferences about system state distinguishes InfoSpect from database-oriented monitoring systems.

We note that in this example, as in most others, a relational schema clearly *could* be extended to deal with the situation. Our point is that for unexpected inconsistencies this must be done after the fact, a difficult operation in databases, especially as important information may have already been thrown away. With our approach there is no need to throw away anything. There is no requirement that the set of facts we have be consistent. It is not even required that we have some *a priori* notion of consistency.

Bind assumptions about state as late as possible

The preceding goals and assumptions lead to perhaps our most important design principle: avoid binding any assumptions about the state of the system until the last possible moment. The flexible knowledge representations allowed by declarative logic languages such as Prolog benefit us greatly in this respect.

The same flexibility that allows us to easily integrate diverse tools also allows us to accept data from the tools "without prejudice", and only later interpret these facts within a particular model of behavior (and check that they are consistent with the model).

This is in contrast to the use of relational databases, where an explicit schema is imposed on observations from the network or system. The process known as "data cleaning" (in data warehousing terminology) when outputs from tools are first fed into the RDBMS has a tendency to throw away precisely the anomalous observations that are most interesting. The use of a schema also makes it hard to evolve the system as new network characteristics become important.

With Prolog, the schema is implicitly present only in the queries and inference rules written alongside the data from network tools, and so can be changed at whim. In fact, we can view the process of making a query as temporarily binding a schema to the data for the duration of the query. This late binding is essential in a highly dynamic and evolving environment.

Of course, the nature of the discovery tools we use also imposes something of an *a priori* schema on the data. Note, however, that this is also the case with systems based on the relational model. Furthermore, by using a diverse array of different discovery tools and delaying the unification of their results until query time, InfoSpect can mitigate the effect of these tool-specific representations of network state.

Finally, we note that the *source* of a particular fact about the system is often as useful as the fact itself in determining state anomalies. In InfoSpect all facts are labeled with where they were obtained. This kind of information is typically discarded in database-oriented systems.

4. Implementation

The InfoSpect prototype consists of a central Prolog *knowledge base*, a set of *driver scripts*, and a corresponding set of ad hoc or off-the-shelf *discovery tools*.

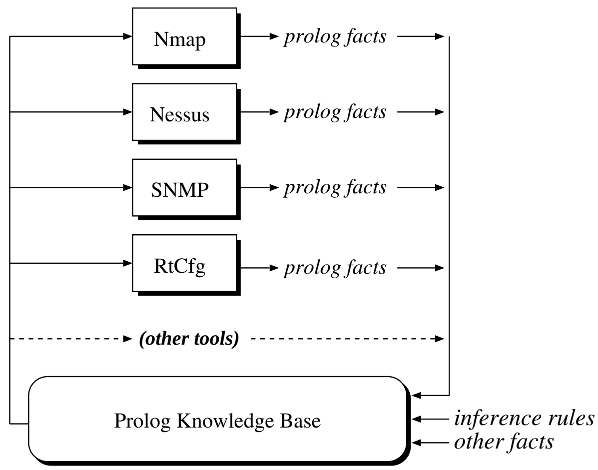

Figure 1. System Data Flow in InfoSpect

Figure 1 illustrates the flow of information through the system. The "main loop" of the system queries the knowledge base for a list of driver scripts to be run, and then invokes each of these (preferably in parallel) to gather information. A driver script queries the knowledge base for information (e.g. the set of possible routers) which it uses to drive a discovery tool. The driver script translates the output of the discovery tool into Prolog facts which are then added to the knowledge base.

This system is highly extensible since it may be updated to use new tools without interrupting normal operation by adding to the knowledge base a set of Prolog facts that describe the set of discovery scripts to be run. These new or modified tools will be run in subsequent iterations.

Entries in the knowledge base come in three flavors:

1. Facts input from external sources (such as a list of well-known Trojan horse ports, or vendor tags for Ethernet MAC addresses);
2. Facts directly observed from some discovery tool (which themselves retain information as to which tool they originate from); and
3. Inference rules which are used to derive new facts from existing ones.

For example, entries of the first kind might include the fact that port 53 is reserved for DNS traffic, entries of the second kind could include which hosts were observed by Nmap to be listening on TCP port 53, and entries of the third kind could include the idea that a machine listening on port 53 is likely to be a DNS server.

4.1. Example tools

Some concrete examples of driver scripts and discovery tools used in our current implementation are:

Network discovery tools: We have a simple SNMP walker written in Python which writes topology and routing information to the knowledge base, and a network consistency checker which queries the knowledge base for likely routers and fetches the running configurations from those routers. The knowledge base includes the router's bootstrap configurations and so the checker can determine what, if anything, has changed.

Our experience has been that automatically interpreting router configurations obtained in the traditional way using `expect` scripts is easier if the information is converted into Prolog as early as possible in the process, and the rest of the job implemented as Prolog rules.

Using this information, predicates can be written to (for example) dynamically check for consistent BGP filters and policies. Another application is determining whether it is possible to transmit an IP packet out of an intranet without traversing a firewall box—a useful feature in a network testing lab.

Host system scanners: Several driver scripts are used for discovering hosts and checking host system security, including the network mapper Nmap [5], and the remote security scanner Nessus [16], fed with the facts derived from the network discovery tools. The results are facts like:

```
  nmap_ipaddr('10.64.201.201').
  nmap_ipaddr('10.64.201.209').
  ...
  nmap_os('10.64.201.201','Solaris 2.6 -
2.7').
  nmap_os('10.64.201.209','Foundry Server-
Iron XL Switch Version 06.0.00T12').
  ...
  nmap_tcp_port_open('10.64.201.201',22).
```

—which indicates among other things that, according to Nmap, a host with IP address 10.64.201.201 exists, and is listening on the ssh port, and probably runs Solaris[1].

From these results, predicates can be derived to locate security anomalies in a variety of end systems. A simple example might be to ask the system for all Windows machines running a vulnerable version of IIS.

These facts are used by the system in other ways as well. A heuristic for discovering routers incorporates operating system information and would include the machine

[1] Nmap includes an operating system fingerprint capability, used here.

10.64.201.209 above as a result. We will see in the next section how heuristics like this can be very concisely expressed.

4.2. A detailed example: DNS walker

In this section we present a much more detailed discussion of one example driver script, to give a more concrete feel for how the system works.

The purpose of the DNS walker is to acquire as many DNS records as possible from as many DNS servers as possible in the local network, and add this information to the knowledge base. As well as requesting zone transfers from DNS servers, the walker repeatedly and recursively makes forward and reverse name lookups for all the host names and addresses it can find. The walker is fairly straightforward, and written in Python; we concentrate here on the Prolog operations related to it.

The walker starts by requesting all values which match the predicate:

```
likely_dns_server(X).
```

This predicate is a pre-defined heuristic for spotting DNS servers, it's simply defined as:

```
likely_dns_server(Machine) :-
    server(Machine,domain).
```

The `server` predicate is a similarly-defined heuristic for spotting servers in general; it's a little more interesting:

```
server(Machine,Svc) :-
    ipservice(Svc,Port,tcp,_),
    nmap_tcp_port_open(Machine,Port).
server(Machine,Svc) :-
    nmap_tcp_port_open(Machine,Svc).
```

This allows us to specify services by name (as in the example above) or port number. The upshot if this is the system regards a machine as a *potential* DNS server if Nmap saw it listening on the domain port.

The level of indirection afforded by the `likely_dns_server` heuristic is important. It gives us a way to flexibly integrate tools without introducing fragile dependencies between them: the DNS walker does not need to be aware of the operating of Nmap, and we could remove Nmap and substitute some other source of port information if we wanted, or indeed do without automatic discovery of DNS servers altogether, and manage with a set of user assertions about which machines should be queried for DNS records.

Conversely, the wrapper for Nmap does not need to concern itself with outputting facts in a form friendly to the DNS walker. The Prolog functions we have listed above give a flavor of the conciseness and flexibility in tool integration that the use of a logic language affords us.

For each potential DNS server, the walker starts from an initial list of host IP addresses, obtained in a similar manner from the results of running previous tools, and performs it's walk over the record space of the server. The output of the walker consists of two types of Prolog facts. The first simply confirms that a DNS server was found at a particular address; it's therefore a stronger statement than `likely_dns_server(X)`. For example:

```
dns_working('10.64.201.201').
dns_working('10.64.209.2').
...
```

The second type of fact indicates the existence of a DNS record of a particular type, in a particular server, with particular name and value. For example:

```
dns_record('10.64.201.201','A',
        'kristeva.smoke.sprintlabs.com.',
        '10.64.202.54').
```

—indicates that the DNS server 10.64.201.201 has a A record specifying that kristeva.smoke.sprintlabs.com has IP address 10.64.202.54. All information is kept, including the address of the server which supplied the record. Even the name of the predicate indicates that the DNS walker, and not some other driver script, was the source of the information.

These facts are now available as input to other discovery tools (for example, a driver script might require a list of machines pointed to by MX records), and can also be queried for inconsistent or anomalous configurations. For instance, the following Prolog function will uncover servers with inconsistent A-records:

```
dns_has_two_arecs(Server,V,N1,N2) :-
        dns_record(Server,'A',N1,V),
        dns_record(Server,'A',N2,V),
        N1 @> N2.
```

More sophisticated queries are also possible. For instance, this Prolog function succeeds if a DNS server has an inconsistent pair of PTR and A records (i.e., forward and reverse address/name bindings):

```
dns_ptr_conflict(Server,Ptr,DNS,A) :-
    dns_record(Server,'A',DNS,A),
    dns_record(Server,'PTR',Ptr,DNS),
    not(ip_arpa_equiv(A,Ptr)).
```

The `ip_arpa_equiv(A,Ptr)` function succeeds if Ptr is the "`.inaddr.arpa`" representation of A (or vice versa).

4.3. Performance

We ran InfoSpect in the Sprint Labs internal network of about 300 hosts, more than 20 of which are IP routers. With our current suite of discovery tools, this results in a knowledge base of around 25,000 entries.

There are two aspects to the performance of the system: time taken to perform a query against the knowledge base, and time take to collect information from the network. Together these determine how up-to-date the information in the knowledge base can be, and how quickly this information can be interpreted to diagnose problems.

Query performance is good: a typical query (for example, to determine routing table inconsistencies) takes only a few milliseconds to execute on a modern workstation. We have not addressed the issue of formulating queries so as to optimize performance at the Prolog level since this has not been a problem so far. Recent work on high-performance declarative languages such as Mercury [10] may help to address this below the level of the language.

Runtime of discovery tools, on the other hand, is dominated by network communication latencies and, more significantly, by the need to throttle some network probes to prevent undue load on the systems being monitored. Many of our driver scripts (such as the port scanners) take on the order of minutes to complete. Consequently, the knowledge base is always a few minutes behind the state of the system. On this basis, InfoSpect does not fall into the "real-time anomaly detection" class of applications, but the speed of queries does allow complex, unanticipated questions to be asked and answered in a timely fashion.

5. Ongoing work

There is much short-term work left to do in InfoSpect. The addition of more discovery tools to the collection we have now will extend the scope of the system but also allow us to gain more experience with constructing the Prolog functions that integrate tools and interpret the knowledge base.

A larger unresolved issue is how best to represent time in InfoSpect: the knowledge base at present holds only observations about the network from the loosely-defined "recent past". A natural first step is to add a timestamp to every fact in the knowledge base which is directly generated from a discovery tool - note that this change can in fact be made to a running InfoSpect system by adding a trivial rule for every class of observation which removes the timestamp. Inference rules and queries could then be formulated over a set of similar facts with different timestamps, but we would prefer a more structured approach to the problem. Two promising avenues to explore are the kind of timing techniques employed in high-level hardware design languages, and temporal logics. A challenge will be to implement a framework for handling time without unduly expanding the state space of facts we have to deal with.

While we are unaware of other work using logic languages to integrate networking monitoring tools, the use of rule- and/or event-based systems in network monitoring is well established. A recent development has been to apply techniques from data mining and machine learning to derive a set of empirical rules about the "normal" behavior of a distributed system or network, and use these in turn to detect anomalies [11]. Such work is complementary to ours and operates at a higher level of abstraction, we speculate that InfoSpect's knowledge base offers a rich foundation on which to build such tools.

6. Conclusion

Network management and configuration is an increasingly important and complex part of any corporation's business. Logic languages appear to offer significant advantages in simplifying the difficult tasks of network and security administration. We have built a prototype system which has been used to monitor the local intranet, and in doing so uncovered hitherto unknown inconsistencies and potential security vulnerabilities.

Our experience so far has been good: wrapping existing network tools so that they are both driven from the contents of the knowledge base and deposit their results into the knowledge base has proved relatively simple. By decoupling our approach from the infrastructure as much as possible, we avoid jeopardizing the dependability of the system we are trying to manage.

Prolog has proved to be highly effective at unifying the results of disparate tools. Furthermore, Prolog queries to uncover inconsistencies in the network state (for example, duplicate or potentially forged DNS records) are remarkably concise and intuitive; typically three or four lines.

We are currently extending the work on router and routing policy configurations to provide the kinds of answers network operators need in the daily running of complex networks and services, including backbone networks.

7. Acknowledgments

We would like to thank John Larson and the Sprint Labs Infrastructure Group for letting us loose on the Lab network in the early stages of this work. We are also grateful to the anonymous reviewers of this paper for several insights and useful suggestions.

References

[1] W. Clocksin and C. Mellish. *Programming in Prolog*. Springer Verlag, 1984.

[2] P. Dourish, D. Swinehart, and M. Theimer. The Doctor Is In: Helping End Users Understand the Health of Distributed Systems. In *Proceedings of the IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, December 2000.

[3] M. Golm and J. Kleinoeder. Ubiquitous Computing and the Need for a New Operating System Architecture. Online at `http://www4.informatik.uni-erlangen.de/Projects/JX /Papers/ubitools01.pdf`, 2001.

[4] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall. A System Architecture for Pervasive Computing. In *Proceedings of the 9th ACM SIGOPS European Workshop, Kolding, Denmark*, pages 177–182, September 2000.

[5] Insecure.org. The Nmap stealth port scanner. `http://insecure.org/nmap/`, 2001.

[6] ISI. RAToolSet. `http://www.isi.edu/ra/RAToolSet/`, 2001.

[7] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. In *Proceedings of ACM SIGCOMM 2000*, pages 175–187, 2000.

[8] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Observations on the dynamic evolution of peer-to-peer networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.

[9] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP Misconfiguration. In *Proceedings of ACM SIGCOMM 2002*, August 2002.

[10] T. Mercury Project. http://www.cs.mu.oz.au/research/mercury, 1996.

[11] M. N. nez, R. Morales, and F. Triguero. Automatic discovery of rules for predicting network management events. *IEEE Journal on Selected Areas in Communications*, 20(4):736–745, May 2002.

[12] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: Accounted Execution of Untrusted Code. In *Proceedings of the fifth Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.

[13] RIPE. RPSL. `http://www.ripe.net/ripencc/pub-services/db/irrtoolset/documentation/`, 2000.

[14] A. Shaikh, L. Kalampoukas, R. Dube, and A. Varma. Routing Stability in Congested Networks: Experimentation and Analysis. In *Proceedings of ACM SIGCOMM 2000*, pages 163–174, 2000.

[15] J. S. Shapiro, S. J. Muir, J. M. Smith, and D. J. Farber. Operating System Support for Active Networks. Technical Report MS-CIS-97-03, University of Pennsylvania, February 1997.

[16] The Nessus Project. `http://www.nessus.org/intro.html`, 2000.

[17] B. Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Cambridge, MA, USA, March 2002.