The Doctor Is In: Helping End Users Understand the Health of Distributed Systems

Paul Dourish[1], Daniel C. Swinehart[2], and Marvin Theimer[3]

[1] Dept Information and Computer Science, University of California Irvine, Irvine, CA 92697
jpd@ics.uci.edu
[2] Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304
swinehart@parc.xerox.com
[3] Microsoft Research, One Microsoft Way, Redmond WA 98052
theimer@microsoft.com

Abstract. Users need know nothing of the internals of distributed applications that are performing well. However, when performance flags or fails, a depiction of system behavior from the user's point of view can be invaluable. We describe a robust architecture and a suite of presentation and visualization tools that give users a clearer view of what is going on.
Keywords: Distributed systems, visualization, diagnostics, management.

1 Introduction

Ideally, users should not have to concern themselves with the internal workings of an application. But most modern personal computer applications are composed from distributed networks of computation, many systems working together to cause the effects the user sees. As complexity and geographic scope increase, it becomes increasingly difficult to answer the questions: "What's happening here?" and "What should or can I do in response?" Current systems typically try to mask faults and performance slowdowns, but leave the user hanging helplessly when they do not fully succeed.

System administrators possess all manner of tools to reveal the behavior of hardware and software systems. The information these tools present is generally focused on the individual components of the system (disks, servers, routers, processes, etc.), and on measures that evaluate the overall health of the environment, rather than end-to-end user applications, distributed services, and the like. Moreover, the information provided by these tools is typically detailed technical information, such as dropped packet counts, throughput characteristics, CPU load and so forth. We believe that the behavioral information that these tools measure could be of considerable value to end users as well. If presented in a focused context and in a way users can comprehend, such information can go far in distributed settings to explain the relationship between system action and the user's immediate tasks.

A. Ambler, S.B. Calo, and G. Kar (Eds.): DSOM 2000, LNCS 1960, pp. 157–168, 2000.

Consider a commonplace example: ordering books online from home. At some point, the process abruptly stops cold. What caused the failure? The local machine, the modem, the phone line, the ISP or backbone router, or the vendor's server? How can an end user understand enough of what is going on to make an informed decision about how to proceed or which customer support line to call? (Depending on the nature of a problem, system administrators may be either unaware of it or unconcerned about the severity of an isolated anomaly, unless alerted by a well-informed user.)

Although particularly noticeable, failures are not necessarily the most valuable behaviors to unravel for users. In particular, properly-functioning but overloaded systems can mimic actual failures, but the most appropriate user response (wait out the rendering of a web page, select a less loaded printer or file mirror site, or simply come back later) is entirely different. If accurate completion estimates can be obtained, a user can make even better decisions. In other words, it is lamentable but true that users continually need to understand system behavior and adjust their own activity to match it. How can we develop systems and tools that can supply these kinds of insights to end users?

This is the question we have addressed in our "Systems Health" project. Its goal is to create an infrastructure for collecting, processing, distributing and presenting to end users information about the ongoing "health" of individual distributed applications: their current status, their overall performance, historical patterns of activity, and their likely future state.

This goal presents a number of challenges. Traditional monitoring tools show "vertical" slices through the system, showing one component in considerable detail and "drilling down" to see more specifics. They collect information that both notifies administrators of an individual problem and provides them with enough information to solve it.

Our concerns are somewhat different. End users need specific, relevant, contextualized information about the components they are actually using and how these components combine to accomplish their tasks. As we have discussed, there may not actually be a "problem" to be solved; the goal may instead be simply to assess the system's state, providing the user with options. For example, a user is likely to be more interested in locating an alternate functioning printer than in knowing how to restore a failed printer to operation.

Making sense of the current state of a system often requires access to historical information. Often, the only way to adequately assess some information is in terms of its variations over time.

In a distributed application, the needed information does not reside in one place. To characterize the behavior of a user's "job" may require gathering remote server performance information, end-to-end network congestion information, and data about local workstation activities, then presenting the information in a form that does not directly match the system's actual structure.

The last thing a user needs is diagnostic tools whose robustness is inferior to the systems they are evaluating. Since many health components will of necessity operate on the same equipment as the observed systems, It is critical that these

components operate with minimal impact on the underlying applications, that they fail significantly less often, and that they continue to operate even when major parts of the system have failed.

In summary, our system must possess a number of characteristics:

1. *User-centered*. Our primary concern is to give information relevant to the needs of a system's end users and expressed in terms they understand.
2. *End-to-end*. Most real work tasks involve a variety of components throughout the system. We must describe all those components acting in concert.
3. *Adjustable*. Users need ways to take different routes through this information, and the system must respond to changes in user concerns. Gathering the needed data should be an automatic consequence of user interest.
4. *Historical*. Since some situations can only be understood in context, we need to be able to provide information about the past as well as the present.
5. *Robust*. The system is intended to help users understand problems that may disturb their work. It is critical that those problems do not also interfere with the functioning of the health system. We refer to this criterion as "fail last/fail least"; it drives a considerable amount of our design.

2 Related Work

Software tools to monitor system and application behavior developed in a rather ad hoc fashion, as commands to extract information collected by the modules that make up a software system. The introduction of the "/proc" virtual filesystem in Eighth Edition UNIX [8] was an early attempt to systematize access to active system information.

SNMP, the Simple Network Management Protocol, introduced a coherent mechanism for structuring and accessing management information across a network [13]. SNMP tools remotely query and control hierarchical stores of management information (Management Information Bases, or MIBs). SNMP underlies many integrated tools for distributed system monitoring and management, such as Unicenter [2]; recent enhancements extend to the inner workings of host systems, services and processes. Similar mechanisms, now operating under the umbrella of the Desktop Management Task Force [3] include Web-Based Enterprise Management, the Desktop Management Interface, and Windows Management Instrumentation. Simpler, if less ambitious, freeware or shareware systems include Big Brother [9] and Pulsar [5], designed to monitor an enterprise and inform administrators of problematic conditions. NetMedic [7] does cater to the end user, using passive analysis of Internet traffic to provide explanations for some performance mysteries of Web browsing.

Planet MBone [10] is a research effort to provide visualizations of diagnostic information, simplifying understanding of the Internet's multicast infrastructure. Similarly, software visualization (e.g. [14]) and algorithm animation [1] prototypes apply graphical techniques to provide diagnostic views of the structure and behavior of software systems.

Dourish [4] has argued the value of visualizations as an aspect of normal interaction with software systems, to provide "accounts" of their operation similar to the physical embodiments of everyday objects. Here, a view into a program's behavior is not provided for debugging, but rather to support the ongoing management of user activity. One implication of this perspective is that a system's account of its current behavior should meaningfully reflect the user's tasks rather the system's structure, echoing our end-to-end "horizontal slice" approach.

3 Architecture

Our health system must satisfy both the interactional needs of the system and the need to provide explanations to end-users. The architecture we have designed to meet these requirements is three-tiered. At the top, a variety of *health applications* both collect and display information about the system's health. In the middle is the *Health Blackboard System*, the abstract model to which applications are written; the blackboard is in turn supported by the *Health Repository*, a storage layer tailored to our specific needs.

3.1 Health Blackboard System

Our architecture begins with a relatively conventional agent/database approach. A common information store, the *blackboard*, acts as an indirect, persistent communication channel among any number of individual processing elements, or agents. Agents produce or consume information by writing to or reading from the blackboard. Our architecture (Fig. 1) is built around three different sorts of agents: *gatherers*, *hunters* and *transducers*.

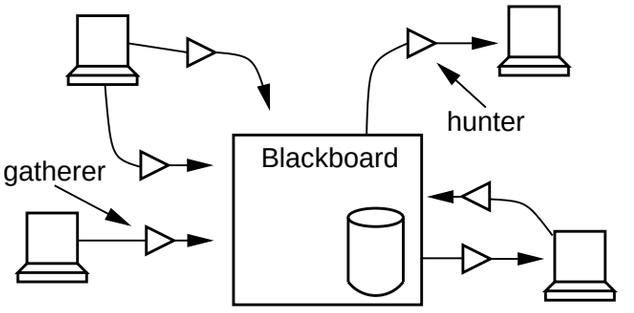

Fig. 1. Hunter and Gatherer Processes Add, Remove Health Data from a Blackboard

Gatherer agents are responsible for gathering different pieces of information from the distributed environment and keeping it up-to-date, by adding or replacing items on the blackboard. Running on or near the computers where the

information is generated, gatherers enter onto the blackboard information from a variety of sources: SNMP interfaces, application management interfaces, or system-specific performance queries, allowing the remainder of the system to be independent of the precise collection mechanisms. Proxy gatherers can be deployed to gather data, using other protocols, from hosts that are not able to fully participate in our architecture, and record the results in the blackboard.

Hunters are the components within presentation applications by which information flows out of the system again. Hunter agents consult the blackboard for health information to process and present in some sort of visualization to a user. They function either by searching explicitly for the desired information or by scheduling notification events for new information matching their interests.

Transducers act both as hunters and as gatherers. A transducer is created to process or summarize raw data into items of greater value to the user. It extracts relevant items from the blackboard, adding new items representing the higher-level account of the system's behavior.

Permanently deployed gatherers monitor specific system components on each participating host, in order to maintain a historical record of base level performance information. Most, however, are created when the information they gather is needed. When a hunter asks for information about, say, a particular server load, a *Gatherer Factory* process on the server host creates a gatherer for that information; the hunter either waits for the first results or schedules future notifications. The blackboard will request the removal of a non-persistent gatherer when there is no longer any hunter interested in that information.

Thus, health data enters the system via gatherers, flows via the blackboard through some number of transducers before playing a part in the reports of one or more hunters. (Data representing the load on a network link might be used to generate reports for the many applications that make use of that particular link.) Multiple timestamped items may be inserted concerning the same measured quantity in order to produce a historical record, where needed. Throughout the process, the blackboard controls the dynamics of the system, presents a uniform model for the collection and management of health information, and connects information providers with information seekers.

Our blackboard is implemented as a tuple-space, not unlike that provided by the widely available JavaSpaces [6]. For consistent interoperation between hunters and gatherers, we employ a conventional structure for tuples. Each data object is accompanied by the host, system, subsystem and component that generated it, a timestamp, a name and optional parameters. The data can be any serializable Java object, permitting the system to store and manage information from a wide range of sources, information whose form may change over time.

As a simple example, consider a hunter that maintains a server's load average graph. The hunter initially indicates its interest in this information by putting onto the blackboard a request tuple of the form `<server1, system, cpu, *, load, *>`, thus registering its interest. Any matching tuple is delivered immediately to the hunter; otherwise, the blackboard directs the *GathererFactory* on *server1* to create a new instance of the *LoadGatherer* class which is regis-

tered as a supplier of `cpu load` data. The new gatherer periodically adds server load records to the blackboard, generating events that will waken any waiting hunters, including the original one, which now can extract the load information and present it to the user. If the LoadGatherer instance cannot directly generate the information, it may serve as a transducer, recursively extracting additional blackboard tuples to produce its result. Should this hunter go away or become unreachable, the blackboard will delete the pattern record. Periodically, another process garbage-collects gatherers whose information is no longer needed.

This example highlights several aspects of our approach. The blackboard separates the collection of information (gatherers) from the routing of information to interested processes (hunters). This indirection allows us to interpose other agents that process low-level records into higher-level information by combining information on the blackboard. It also allows us to limit the network traffic and processor load involved in collecting and collating information. Finally, health system components persist in the system only while they are needed.

We have stated that historical information is often of considerable importance in understanding the state of a system. Retaining data records for extended periods can provide the same vital function that conventional system logs achieve, with the added advantage that the resulting tuples can be organized through various structured queries. In our current prototype, historical records can be retained by adding multiple timestamped tuples matching the same search specification to the blackboard. Further, through an extensible "hints" mechanism, clients can specify degrees of liveness of the information they require, "time to live" hints for information that is added, and so forth. While this does not yet address all the historical requirements, it provides enough flexibility to allow basic management of the temporality of near-live data.

3.2 Health Repository

Although the Blackboard abstraction can be supported by very simple, centralized, in-memory information stores, our robustness requirements create additional criteria for a storage layer. First, health information must be available with low overhead. Since not only error states but also ongoing performance information must be presented to the user on a continuing basis, we must avoid server bottlenecks and the overhead of synchronous network requests to fetch and store data. Second, the health system must not itself appear to fail as a consequence of other problems. For example, in the case of a network partition, the system must continue to function as best it can. Third, logging requirements argue for storage of diagnostic information beyond the capacities of individual workstations or application servers. Finally, health reports from a large enterprise could overwhelm a centralized server.

These criteria suggest the use of a replicated data store. By storing health information in persistent, replicated repositories, gatherers may collect and report their findings to local replicas, confident that the data will eventually reach those that need it, even in the face of network congestion or partitioning. Similarly, hunters can report the latest information they were able to receive; they can use

discontinuities in timestamps to identify the approximate point of failure along a broken network path. We can present the appearance of a system with very low probability of total failure.

But a fully replicated store will also not scale. This argues for partial replication, where information is replicated only where needed, according to the patterns of information requests. Ideally, local replicas should be readily available both to the gatherers that create the information and to the hunters that use it. Furthermore, for information that is of longer term value, long-lived replicas should be retained by servers charged with maintaining historical data. Additional replicas can be designated for further protection against information loss. This approach implies that some amount of data inconsistency can be tolerated, if necessary, in order to keep functioning. For system health information, we need data to be accurate when possible, but slightly out of date information is better than none and can be tolerated when the latest values are unavailable.

Based on these considerations, we chose to adapt the Bayou system [17] as our repository layer. Bayou is a weakly-consistent replicated database, developed originally to support ubiquitous computing environments, where mobile devices routinely experience unpredictable connectivity. Bayou provides mechanisms for dealing with network disconnection and carrying on processing, including *session guarantees* (customizable degrees of consistency management) and *mergeprocs* (application specific code to resolve conflicts after reconnection) [16].

Bayou's replication supports our requirements for high availability. Its weak consistency adequately supports continued operation in the face of temporary failures and network partitions. However, its fully general algorithm for resolving update conflicts is more heavyweight than we need. Health information has specific dynamic patterns of information generation and modification, which we can exploit in designing a replication update scheme. We developed a variant of the original implementation, enhanced for rapid (sub-second) dissemination of updates to all replicas, to underlie our blackboard abstraction.

3.3 Infrastructure Implementation

We have built both sorts of gatherers. Simple permanent gatherers use system tools such as SNMP probes or such as the Unix commands *vmstat*, *iostat*, *netstat* or *ps* to extract host-level statistics. More sophisticated ones address common concerns. For instance, we use a version of the Unix *df* command, modified to report information about NFS connections with gingerly probes that do not cause an NFS wait if a server is dead. We have also built very specific gatherers, launched as needed to monitor locally produced applications, such as a large infrastructure for managing scanned documents.

The Health System is written almost entirely in Java. Added to the approximately 12000 lines of Java code, a few thousand lines of C is used in data extractors. The system employs both a basic centralized repository for testing purposes, and our Bayou-based replicated repository. The Bayou port, written in java, uses MySQL as its relational database component. A designed approach to partial replication awaits implementation.

4 A Web-Based Approach: Checkup

We turn now to a description of the most extensive application we have developed to demonstrate the health architecture, a web application called *Checkup*. Checkup provides a coherent interface to a wide range of information, replacing the bag of complex software tools that a user might employ on different platforms to determine what is going on.

Figure 2 excerpts snippets from four dynamically-generated Checkup pages[1], beginning with the root page for the host AlicesHost, then progressing through the (bold-faced) links to additional views. The root page (Fig. 2, panel 1) is a top-level view of that host's overall state, wrapping the output of such standard Unix performance tools as *pstat*, *vmstat*, and *fs* into HTML presentations. This page includes links that, when invoked dynamically, produce more detailed performance analyses. For instance, following the "mounted file server" links will dynamically create root pages for those related hosts. Similarly, clicking a "per-process" link creates the page of (Fig. 2, panel 2), a table listing attributes of all running processes sorted into a user-specified order. A link from each process yields a detailed account of that process's performance and resource use: names, open local and remote files, and open network connections (not shown). Finally, one can invoke a net path analysis tool (Fig. 2, panel 3), by clicking on one of the open connections. This tool encapsulates many behavioral parameters that the user would otherwise need to supply by hand, then combines the results from both hosts into a presentation that is far more comprehensible than direct operation of the underlying UNIX tool.

This approach has several advantages over previous tools. The use of HTML hyperlinks to annotate the output of standard monitoring commands means that the Web pages define a space of information with a high degree of branching, encouraging the user to explore specific features of interest, rather than building up a general picture. The path that a user follows to a particular leaf node can contextualize the way in which the tool appearing there is used. Finally, a serendipitous advantage is that Checkup pages can readily be created with links to other web-based diagnostic, monitoring, and repair tools, extending the scope of the system with little effort.

4.1 Checkup Implementation

Checkup (see Fig. 3) is based on Java servlets, server-based objects that create Checkup pages on demand as links are followed, through the auspices of a mechanism similar to Java Server Pages [15]. To the Health Blackboard, servlets are hunters, collecting information from the blackboard to be reported back to the user. As usual, these hunter requests lead to the on-demand launch of the necessary gatherer agents, whose results are posted to satisfy the user requests. The implementation includes mechanisms for retaining (for a brief time within a

[1] Full-resolution color versions of all user examples are available at `http://www.parc.xerox.com/csl/projects/systemhealth/md/examples`.

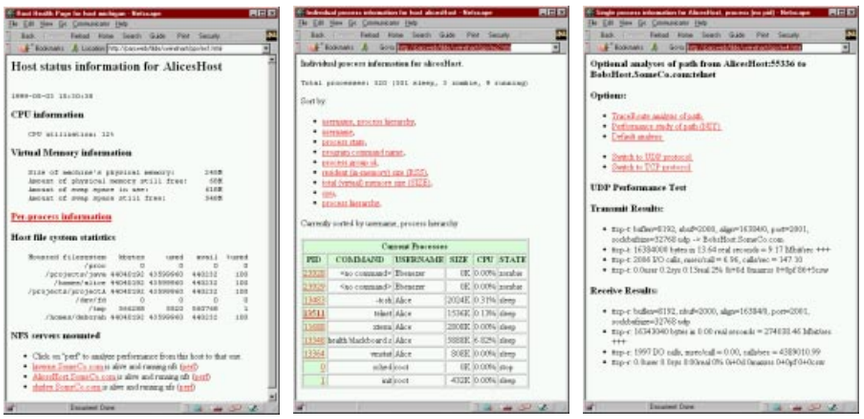

Fig. 2. A Sequence of Checkup Web Pages, Generated by Servlet Hunters.

session) time-consuming computations, such as the top-level attributes of all of a host's processes. Servlets that create related pages can use such cached information to increase responsiveness. Some pages, such as the one depicted in Fig. 2 panel 3, issue multiple requests for information provided by multiple hosts.

Servlets annotate their web pages with dynamically computed information and with hyperlinks to more detailed analyses. For example, the root page servlet checks what other systems a host depends on for its file service, and for each, either indicates that the corresponding server is not running or provides a link to the root page for that server.

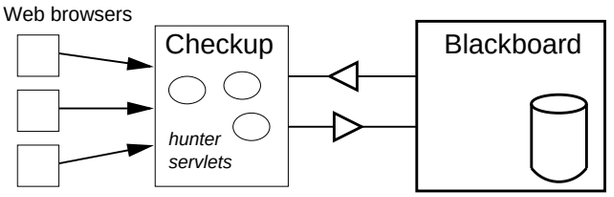

Fig. 3. Checkup Is a Servlet-Based Health Application for Standard Web Browsers

4.2 Application Helpers

We can use the same infrastructure both to evaluate and inform a specific application, by linking the application to the health system via a small "Health Helper" program.

As an example, we have built a Health-enabled variant of *emacs*, a text editor available on many platforms. We chose *emacs* because it is widely used, dependent on many components in the environment, and highly customizable through provided APIs. When *emacs* is started, a small "emacs helper" is also started to provide bidirectional communications between *emacs* and the blackboard. Acting as a gatherer, this program posts specific performance information, including details of memory and CPU usage that other applications can use to help them coexist with this large, cycle-hungry application. Acting as a hunter, the helper can supply *emacs* with information that can improve its robustness. For example, *emacs* can divert an attempt to download library code from an unresponsive server to one that is available; similarly, *emacs* can determine that there is insufficient room to save the next version in the current location, in time to prevent an attempt that could result in failure or lost work.

A Checkup page could be created to present to the user the detailed *emacs* performance information provided by the helper program. Thus, users do not have to learn a new application to monitor this application, but can use methods they are already familiar with.

5 Visualizing System State

Finally, we have been exploring the presentation of system health information through visualization tools. Visualization enables the movement of activity from the cognitive system to the perceptual system [11], making patterns and correlations in the blackboard data directly perceptible. Visualization through animation of system data [12] directly supports our intuition that patterns of significance to the user occur all the time, not only in the face of failure. Since we assert that systems health depends both on individual components and on the interactions and relationships between them, we believe a successful approach will convey these relationships as well as the more objective data. Further, we seek a presentation that minimizes the technical knowledge required to interpret it, which argues for a perceptual rather than a cognitive approach.

We have prototyped a series of simple visualizations. The *Ping Radar* visualizes the performance of a network, as seen from a particular host. In Fig. 4, columns represent hosts. Rows are snapshots at successive instants. The color of a cell estimates a round trip time to that host. The radar "beam" replaces rows cyclically at fixed intervals, giving us a picture of recent activity. Over time, patterns reveal the reliability of each connection, periods of network downtime, etc. We can see that one host appears to be down (black column), that hosts on the left are closer to us than those on the right, and that network response to distant hosts is quite variable.

Similarly, we built a *Process Map* that depicts the processes on a single host, using the position and color of data boxes to represent the memory each has consumed, their run times, and their current states. Readily perceived separations emerge between temporary small processes (simple commands), long-running processes (servers) and anomalous processes (runaway computations).

These tools are blackboard clients, permitting exploration of different ways to organize and visualize relevant information.

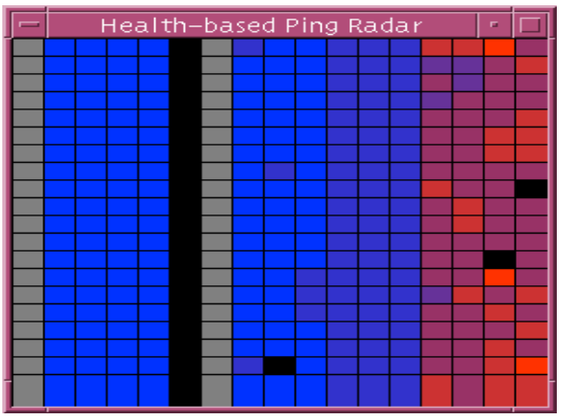

Fig. 4. The *Ping Radar* Gives a Continually Updated View of the Recent Network State

It is clear from these early explorations that the context in which information is presented is of considerable importance. Our examples so far all deal out *absolute* data: ping times, absolute memory consumption, etc. Although these readings provide valuable information, there are times when relative values, such as deviations from the norm, anomalous readings, or the differences between readings, would provide more insight.

6 Conclusions

Traditional diagnostic tools are optimized for system managers and administrators. We believe that information about the health of networked distributed systems can be of equal value to end users, if properly presented. We have the beginnings of a tool set that can tell users what is going on with their applications. We employ a blackboard architecture and multiple cooperating agents to build up information from "vertical" system views into "horizontal" slices corresponding to the end-to-end structure of user activities, combining disparate information sources into a coherent user report. This architecture is effective both for creating expressly designed health views and for augmenting the capabilities of existing applications to explain their behavior.

Basing our repository on a weakly consistent replication model enables a high degree of fault-tolerance in support of our "fail least/fail last" criterion, maximizing the availability of correct and at least partially complete information.

Checkup and our visualizers are only a first step in building health applications. Considerable work remains, particularly in the exploitation of historical

information and in the ability to report coherently about application behavior. The infrastructure we have developed is a basis for further investigation.

Acknowledgments

Mark Spiteri and Karin Petersen built the Bayou-based blackboard repository. We would like to thank Anthony Joseph, Ron Frederick, and Bill Fenner for their advice and assistance, and the reviewers for many useful insights.

References

1. Brown, M. and Najork, M.: Algorithm Animation using 3D Interactive Graphics. *Proc. Sixth ACM Symp. User Interface Software and Tech.* Atlanta (1993) 93-100
2. Unicenter TNG Product Description. Computer Associates International, Inc., Islandia, NY 11788 (1998) URL: `http://www.cai.com/products/unicenter`
3. Distributed Management Task Force (2000) URL: `http://www.dmtf.org/`
4. Dourish, P.: Accounting for System Behaviour: Representation, Reflection and Resourceful Actions. In Kyng and Mattiassen (eds), *Computers and Design in Context*, Cambridge: MIT Press (1997)
5. Finkel, R.: Pulsar: An Extensible Tool for Monitoring Large UNIX Sites. *Software Practice and Experience*, 27(10) (1997) 1163-1176
6. Freeman, E., Hupfer, S., and Arnold, K.:*JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, Reading MA 01867 (1999) 344 pp. ISBN 0-201-30955-6
7. Net.Medic: Your Remedy for Poor Online Performance. Lucent Network Care, Sunnyvale, CA 94089, (2000)
   URL: `http://www.lucent-networkcare.com/software/medic/datasheet/`
8. Killian, T.: Processes as Files. *Proc. USENIX Summer Conf.* Salt Lake City (1984) 203-207
9. Big Brother: Monitoring and Notification for Systems and Networks. The MacLawran Group., Montreal (1998) URL: `http://maclawran.ca`
10. Munzer, T., Hoffman, E., Claffy, K., and Fenner, B.: Visualizing the Global Topology of the MBone. *Proc. IEEE Symp. Info. Vis.* San Francisco (1996) 85-92
11. Robertson, G., Card, S., and Mackinlay, J.: The Cognitive Coprocessor Architecture for Interactive User Interfaces. *Proc. ACM SIGGRAPH Symp. on User Interface Software and Tech* (1989) 10-18
12. Robertson, G., Card, S., and Mackinlay, J.: Information Visualization using 3D Interactive Animation. *Comm. ACM*, 36(4), (1993) 56-71
13. Schoffstall, M., Fedor, M., Davin, J., and Case, J. A.: Simple Network Management Protocol. RFC 1098, SRI Network Info. Ctr., Menlo Park (1989)
14. Stasko, J., Domingue, J., Brown, M. and Price, B.: Software Visualization: Programming as a Multimedia Experience. Cambridge MA MIT Press (1998)
15. *Java Server Pages Technical Specification.* Sun Microsystems, Palo Alto, CA (2000) URL: `http://www.javasoft.com/products/jsp/`
16. Terry, D., Demers, A., Petersen, K., Spreitzer, M., Theimer, M. and Welch, B.: Session Guarantees for Weakly Consistent Replicated Data. *Proc. IEEE Intl. Conf. Parallel and Distributed Info. Syst.* Austin (1994) 140-149
17. Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C.: Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. *Proc. ACM Symp. on Oper. Syst. Principles* (1995) 172-182