The Case for Predicate-Oriented Debugging of Sensornets

Arsalan Tavakoli
U. of California - Berkeley
Berkeley, CA 94720
arsalan@cs.berkeley.edu

David Culler
U. of California - Berkeley
Berkeley, CA 94720
culler@cs.berkeley.edu

Scott Shenker
U. of California - Berkeley
Berkeley, CA 94720
shenker@icsi.berkeley.edu

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Metrics—*Debugging aids, Distributed Debugging*

General Terms

Management, Reliability, Verification

Keywords

Predicates, Visibility, Scheme, Interpreter

1 Introduction

In sensornets, deployment is a three-stage process: Design → Testbed → Real-World. They say the difference between theory and practice is that in theory there isn't any, but in practice there is. Well, the same applies to the difference between testbeds and the real-world; in theory there shouldn't be any, but in reality there is. Sensornet literature is replete with evidence of deployments in which results did not meet expectations, either with reduced data yield or shortened network lifetime. Testbed environments are controlled, allow for reprogramming/rebooting of motes, and have a wealth of resources in terms of backchannel, connectivity, and power. Troubleshooting systems is not easy, but one does not have to cope with resource limits to do so.

Figuring out why a system is underperforming in the real world is much more difficult, as many testbed capabilities are stripped away, and without any transparency and accessibility the network crashes. The purpose of this work is to build a system to help us figure this out in real deployments.

Why do testbed results mislead? This is largely because of the sensitivity of sensor networks to their external conditions, whether wireless interference, or temperature, or topology layout. Applications have many implicit assumptions built in, about these conditions and their effects. We want a system where users can test whether these assumptions hold, in an effort to locate where reality has diverged from the testbed system.

Our work focuses on predicate-oriented debugging, as we postulate that erratic behavior is the result of one of these fundamental assumptions being violated. Ideally, we would like a system that could analyze code and provide a set of formal invariants and tell us at any point what is going wrong, but this is extremely difficult, if not impossible, particularly with constrained resources and distributed state. Instead we use predicates as informal predictions, similar to the way a developer uses `printf's` to ensure certain conditions hold. However, rather than instrumenting at single points, developers can specify conditions over system state, local or distributed, and define system action depending on the status of such conditions.

At a high-level, our approach consists of three steps, performed iteratively throughout network lifetime: Specify predicate, monitor predicate, and assess underlying cause when predicate fails.

Our contribution is two fold:

- A minimal-intrusion dynamic instrumentation mechanism that works within the limited constraints of sensornets.
- Enabling monitoring and processing of distributed state by using packets as an enabler of computation and carrier of state.

2 Building Blocks

We do not claim novelty in our work; we have drawn inspiration from work across a variety of domains. Rather, we believe the contribution of our work to be the synthesis of these ideas to address an outstanding need, and overcoming the challenges posed in designing such a system for our resource-starved target environment.

For the sake of brevity we do not list all individual pieces of related work, but rather provide a list of related categories and example systems for each.

Visibility and Dynamic Modification: Marionette [1] provides the end user with the ability to query the value of system variables, and remotely call specially tagged functions. Clairvoyant [2] provides GDB-like capabilities such as breakpoints, watchpoints, and step-by-step execution. Both focus on only a single node at any given point.

Predicates and Filtered Probes: DTrace [3] is a dynamic instrumentation tool for Solaris workstations. It provides an extensive set of probes throughout the entire system,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
HotEmNets'08, June 2-3, 2008, Charlottesville, VA.
Copyright 2008 ACM 978-1-60558-209-2/08/0006 ...$5.00

activated by custom user scripts. While these probes are inactive, they do not affect the functioning of the system in any way. However, the system is designed for only a single machine, makes extensive use of storage buffers, and also allows the state for all instrumentation probes (over 30,000 for even the smallest systems) to be managed by the system concurrently.

Distributed Predicates $\approx$ Network Path: XTrace [4] provides tracing support for networked applications in which a packet's path across different layers in a single machine *and* across different nodes in the network is traced. Processes and applications at each layer must be modified to enable proper logging of packet appearences (and subsequent retrieval) in order to be able to construct a full packet trace at each given point.

Code Capsules and Triggers: Active networking [5] allows packets to carry pieces of code that are executed by routers and other middleboxes in the network. Mate [6] shifted the paradigm of sensornet application development from a compiled one to that of a virtual machine bytecode interpreter. Mate served as the only system application/service, and focused on providing safety, isolation, and resource conflict guarantees to the various code capsules loaded. In contrast, our system must be lightweight, and interact (and provide visibility into the state of) all other existing protocols and applications that composed the system.

3 Predicate Examples

Predicates form the foundation of our approach:

> A predicate is defined as a boolean assertion on whether an entity, either a local variable or an aggregation of state across nodes, meets a specified condition.

In order to provide predicate examples, we examined previous deployments and their failures. In certain cases, the cause of the failures was later determined (typically through extensive post-mortem analysis and ad-hoc online mechanisms), and so we highlight predicates that could have helped narrow down (if not determine) the root cause of the system failures. We specifically chose examples that demonstrate a scenario in which only a certain environmental condition, coupled with natural distributed systems interactions, caused the failure to manifest, making it very difficult to detect in a testbed or with single node-oriented tools.

PEG (Pursuer Evader Games) [7], is a vehicle tracking system in which sensor nodes use a magnetometer to determine the presence of foreign entities. An estimate of the object's position is calculated and relayed to the *pursuer*.

To reduce aggregate bandwidth, each node shares its readings with all neighbors within radio range, and a node deems itself a *leader* if it has the highest reading of all its neighbors. Only the leader estimates the position of the evader and sends a message to the pursuer. One problem with this protocol is that the message containing the sensor reading of the leader node can be lost, causing one more other nodes to also elect themselves as the leader. Having multiple leaders can cause a number of problems, such as redundant or conflicting messages being sent to the pursuer. It could also cause problems when trying to perform *handoffs* between subsequent leaders. This bug is a result of asymmetric packet loss, and would be impossible to detect in a simulation that does not model this aspect of the physical world. A predicate checking for the existence of multiple leaders could be injected, directing all nodes that receive or overhear leader-sourced messages from *more than one* node in a limited time window to trigger an alert.

During one deployment, versioning protocols were used to disseminate data. These systems use sequence numbers, with a valid range of 0-255 and wrap-around accepted, and a node views all sequence numbers that are within 85 (1/3 of the window) of its own to be fresher data. When nodes rebooted (which was a common occurence since researchers overestimated the amount of solar energy harvested), they advertised with a sequence number of 0, leading existing nodes to assume this was fresh data and hence trigger a flood of requests. As nodes periodically rebooted, certain hotspots in the network with heavy traffic and congestion formed, causing application data to be lost. Our predicate would be double pronged: nodes would be programmed to detect irregular jumps in version number (defined as a jump beyond a certain threshold), and also to inform neighbors when rebooting. Nodes could then correlate the two events and issue an alert, allowing an administrator to distribute a fix.

In a problem that was seen in numerous deployments, routing algorithms would crash and fall apart in real-world deployments after performing well on the testbed, often devasting data yield. Careful analysis revealed that a change in topology led to higher density in the network, leading to certain bottleneck nodes having their queues overflow. Since the routing protocols are often reliable protocols, they performed numerous retransmissions, further congesting the network. At this point, a predicate checking the neighborhood density of bottleneck nodes, or one that monitored queue lengths could have provided some insight into the problem.

A similar problem was observed in another deployment, except rather than queue overflows, certain links temporarily degraded in quality. A periodic predicate that calculated the total number of transmissions to send a packet to the base station, along with RSSI measurements from each hop, would have allowed end-users to notice the deterioration of links.

4 System Overview

Our system provides predicate specification, global state monitoring, and dynamic tool deployment. The foundation of our system is Wringer, a lightweight rapid-prototyping framework for sensornets. Our system is designed for TinyOS 2.0 and our discussions assume this operating environment; however, these same principles can easily be exported to other systems.

4.1 Wringer

TinyOS and the more popular mote platforms provide us with the following environment: Static memory allocation, no dynamic loading/unloading of code, 48-60K of ROM, and 4-10K of RAM. Our system must provide dynamic mechanisms to alleviate the first two constraints, and be lightweight as the bulk of the latter two resources would be reserved for the actual system application.

Wringer is at its core a lightweight Scheme [8] interpreter.

We chose Scheme because its concise grammar allowed for a tight implementation, while its expressivity did not limit the user. The system does not provide the full grammar as dictated by [8] due to space constraints, but does provide the core components. One of the beauties of Scheme is that if the basic set of primitives is provided, the rest of the grammar can be defined in Scheme itself.

At a high level, Wringer is composed of three building blocks: primitives, variables, and triggers. Primitives and variables are either hard-wired (*native primitives and variables*), or dynamically loaded (*user-defined*). Primitives serve as event handlers, executed by other primitives, triggers, and incoming packets. Variables provide a mechanism for the user to provide information to Wringer, and for primitives to maintain state. We will discuss triggers shortly.

All *native primitives* are placed into ROM, while *user-defined entities* (including defining other aspects of the Scheme grammar), are put in RAM. While the core primitives must be installed as native primitives, all other primitives are optional, both in presence and implementation style. Although native primitives are faster than interpreted ones, we do not imagine speed as being a driving decision. Rather, native primitives conserve RAM space, while user-defined primitives can be loaded/unloaded at anytime.

At compile time, Wringer obtains a chunk of memory from the operating system (size is user defined), and then implements its own dynamic allocation service to distribute this memory as needed. This includes providing a heap to store defintions/values of user-defined functions and variables, as well as a stack to hold operands as needed during function calls. Modules can be dynamically *unloaded*, alleviating resource constraints.

Wringer functions as a stand-alone application/service in the system stack, with no explicit interaction with other applications. In typical cases, Wringer will make use of a pre-existing routing protocol, such as CTP, and a multi-version data dissemination protocol, such as Drip. However, the only fundamental requirement that Wringer has is access to a radio MAC layer, as the routing and data dissemination protocols can be defined using Wringer itself. This might seem impractical and awkward, but is critical in scenarios where the failure of the routing algorithm or dissemination protocol that renders them unusable (two of our examples from section 3), is in fact what we are diagnosing.

In section 1, we referred to our contribution as a *minimal-intrusion dynamic instrumentation framework*. We have acheived this goal thus far by installing Wringer as a separate application in the system, and if no code modules are enabled (i.e. loaded), then the native application is completely unaffected (except for the occupied ROM/RAM). However, this aspect does not define the interaction and instrumentation component of Wringer.

Similar to Marionette's [1] approach, at compile time, Wringer combs through an application's symbol table to gather the (name, memory-address) mapping for each variable (including those used by Wringer itself). Wringer maintains this information on the PC side, and subsequently allows users to reference variables by their semantic names when writing Scheme code using the notation: *Component.Variable*, i.e. **RadioCountToLedsC.counter**. When sending this instruction to the network, Wringer converts the name into a reference to the appropriate memory address.

Triggers are an integral part of our framework, yet continuously polling variables is an expensive and impractical method for discovering change. Rather we use binary rewriting to allow watchpoints to be inserted in the code. However, unlike Clairvoyant [2], which implements a conditional breakpoint for each instruction to obtain *watch* capabilities, we perform a static analysis on the PC-side, determining all instructions where the variable could be explicitly modified, and only set up breakpoints for each of these instructions.[1] This provides a much more efficient watchpoint; however, since we only catch explicit references we will miss any pointer-based modifications. Future work involves rectifying this issue.

One of the key benefits of native primitives, similar to the use of native predicates in DSN [9], is that they enable extensibility using a clean, standard interface. For example, if a particular application is deployed on a node with a hardware energy-monitoring chip, and a software energy-management component, Wringer could expose this functionality using a parameterized native primitive (e.g. *energyManager()*). The user could then write energy-dependent predicates, such as setting a low-energy alert threshold.

4.2 Additional Mechanisms

Predicate specification consists of three components: trigger(s), condition verifier, and data processor. The trigger invokes the condition verifier, and can be a watchpoint for one or more variables, a timer firing, a function call, or packet reception. The condition verifier is a boolean expression representing the predicate. If the condition verifier succeeds, the data processor executes, which can be local processing and/or packet transmission.

A distributed predicate is still a boolean assertion that is verified or monitored locally by a node or a set of nodes, and so functionally similar to a local predicate. The key difference is that Wringer provides the tools for a single node to utilize state from other nodes, and selected information about network traffic (such as RSSI, packet reception ratio, and link asymmetries), as well as its own state. The gathering and transferring of data is dictated as part of the predicate specifications. For example, if a user wants to determine the number of direct children a node in a collection tree has, Wringer provides mechanisms to dictate that the node cache all the unique sources of received messages over a specified time window, and subsequently report this information to the base station. In essence, the goal of Wringer is to provide a platform for rapid and dynamic implementation of distributed algorithms aimed at providing visibility into the network.

One other notion that Wringer introduces is the concept of *collaborative logging*, which provides replication of node state. A node can either request this service of its neighbors,

[1] Binary rewriting can potentially result in the shifting of memory locations of variables addressed by existing predicates. This will be caught on the PC-side, and will result in updates being issued for existing predicates.

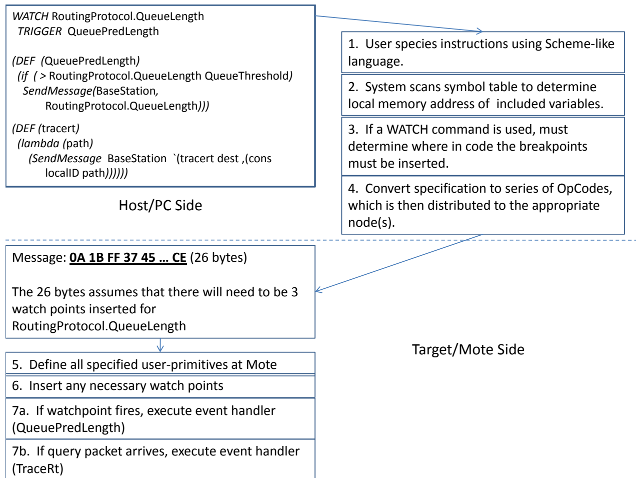

Figure 1. Example Wringer Predicate Deployment

an explicit command can be issued from the user, or it can be triggered by a predicate on a remote node. Examples of this may include maintaining timestamps of the last beacon heard or the RSSI of the most recently received packet on a per neighbor basis.

This information can be used to respond to queries when the *queryHelp*[2] flag is set. Consider the following scenario: No packets have been received from a particular node, and so the user submits a *ping request* with the *queryHelp* flag disabled. No response is received and so the query is resent, with the flag set. Because of the broadcast nature of the medium, the target node's neighbors will also receive the query, and respond with any previously logged information. Thus, the end-user gains insight into whether the node has failed (no recent beacon has been received and the last RSSI was well over the threshold), or is only temporarily unreachable due to poor link quality (the RSSI of the last received beacon narrowly cleared the required SINR threshold).

The final component of Wringer is rapid-prototyping of tools, the third step in our debugging approach. Predicate violation does not provide a root cause analysis; rather it specifies a precise symptom that helps narrow the range of causes. The end-user can then deploy tools which have a greater probability of determining the true cause, such as *ping* and *tracert* when connectivity issues arise. The expressivity of Scheme and the visibility into system state provided by Wringer imply that space constraints are the main limiting factors to tool design.

4.3 Example Walkthroughs

Figure 1 demonstrates the functionality of Wringer by walking through one of our motivating examples. In this case, we are checking to make sure that the queue length does not exceed a certain threshold, as well as enabling *tracert* capabilities. The process begins with the user providing a *definition specification*, using a modified Scheme grammar. The file has three parts: the declaration of the watchpoint and its associated event handler, and also the *QueuePredLength* and *TraceRt* primitives. *SendMessage* and *localID* are a native primitive and variable, respectively. The former provides message transmission capabilities while the latter is the node address. *QueueThreshold* is a variable that we assume was specified previously.

[2]The *queryHelp* flag being set indicates to neighboring nodes that they can respond to a query destined for another node if they possess the ability to do so.

Activity	Metric	Result
Transmitting Snippet	# of Packets	1
	Packet Size	26 Bytes
Inserting Snippet	ROM Space	None
Adding TraceRt	RAM Space	10 Bytes
Adding QueuePredLength	RAM Space	11 Bytes
TraceRt Response	Packet Size	1+2*Hops
QueuePredLength Response	Packet Size	2 Bytes

Figure 2. Footprint Evaulation

On the PC-side, after receiving the specification, Wringer first converts all references to native mote variables to their proper memory address, using the results of its original symbol table scan. It then converts the specification to a series of OPCodes that can be understood by the mote-side interpreter. For this particular specification, the Wringer-specific payload will be 26 bytes (which does assume that the *watch* requires 3 breakpoint insertions.) For brevity we do not describe this conversion in detail, but for example, the conditional *if* statement in *QueuePredLength* requires 6 bytes:

- 1 byte for the OPCode for the *if* primitive
- 1 byte for the OPCode for the $>$ primitive
- 3 bytes for the reference to the native variable
- 1 byte for the OPCode for the user-defined variable

Upon reception, the mote defines the event-handlers, using the OPCodes given to it by the PC-side (Wringer maintains a list of all the OPCodes in use by the motes and maintains (name $\rightarrow$ OPCode) mappings for all of them), sets the watchpoints. At any point, if a watchpoint is triggered, or an incoming packet calls an event handler, the appropriate processing is done, both leading to message transmissions.

Figure 2 provides a breakdown of the cost of implementing this particular specification. The size of the entire specification is 26 bytes, which can fit into a single packet (depending on what packet size is used). The definitions are all done dynamically, meaning no ROM space is utilized. Each primitive definition simply maintains the OPCodes provided in the specification, allowing for tight implementations. We note that all these figures assume the Wringer base framework has already been installed on the mote.

4.4 Current Status and Future Work

Development of an intial Wringer prototype is in progress, designed for the MicaZ, TelosB, and Epic (a new UC Berkeley mote) platforms running TinyOS 2.0. The default front-end is a command-line java tool, but we are also designing a web-based GUI that is part of a larger web-based programming framework.

The testbed is where most of the initial testing will occur. Even in cases where applications are not having any significant failures, the capability of Wringer to provide visibility into the inner workings of the network can be utilized. Another benefit of this phase is compiling a library of *practical*

predicates for different classes of applications/environment, composed of tools that proved most pragmatic and effective in initial testing. For example, collection-oriented applications would likely use predicates that try to identify bottleneck links and connectivity issues, while event-detection applications often require predicates that detect duplicate leaders, and ensure low-energy nodes do not become leaders.

Determining the precise processing overhead of Wringer will be critical. The system is designed to give priority to the native application whenever possible, predominantly by doing the bulk of its processing in non-interrupting and preemptible tasks, rather than commands and events. However, triggers still interrupt program execution, no matter how minor, and raise the possibility of heisenbugs. Identifying such bugs is difficult, but Wringer can provide visibility into its own state in the same manner it does for application state.

Another metric is the scalability of Wringer in larger networks. Predicate specification and on-node processing are unaffected; rather the component that is sensitive to scaling is the underlying routing protocol, which is responsible for distributing predicates, routing responses, and facilitating inter-node communication. As such, we are concurrently testing existing protocols and working on designing a custom protocol that supports anycast, multicast, and unicast in large networks.

The ultimate goal is to bundle Wringer as part of real-world deployments in order to enable visibility into and understanding of network behavior. Highlighting predicates for past deployments retrospectively is a step in the right direction, but true validation of this work can only be garnered through interaction with new failures and scenarios.

5 Related Work

In section 2 we discussed previous work that has served as building blocks for the design of our system. There have also certainly been systems that take different approaches to debugging and troubleshooting.

Sympathy [10] equips the network with high-powered nodes that use eavesdropped packets to determine potential network issues, using a decision tree created by domain experts. Visibility [11] builds on this decision-tree oriented approach, assigning costs to each branch of the tree, and uses this information to design more *transparent* network protocols that are less costly to debug. Both are limited to generic high-level observations.

LibLog [12] instruments systems using low-level kernel libraries that allows for all system events to be logged, and these logs are collected at a central repository, where they are merged together to allow for deterministic replay. Friday [13] allows global assertions to be made across the entire network during these replays. These systems only allow for post-mortem, rather than on-line analysis, and log extensive amounts of data. EnviroLog [14] provides a similar, yet more limited, system that enables replay of sensing information to allow replication of infrequent data gathering activities.

Much work in the parallel and grid computing spaces has been done in debugging. These tools are event-driven [15] with centralized collection, or predicate-oriented with either centralized [16] or hierarchical-decentralized [17] processing. Event-driven systems use kernel instrumentation to log all events performed by the system, and these logs are collected by a central server to allow for post-mortem replay. Predicate-oriented systems allow for individual systems to be instrumented, and then collected at a central server. Aggregations are performed by this central server, except in hiearchical systems, in-network backup servers perform partial aggregations, reducing the load on the network. Yet, the majority of these systems require significant state management, have a great deal of network overhead (with little loss), and are limited to static algebraic predicates with no in-network processing capabilities.

6 Conclusion

The desirability of increased transparency and visibility through predicate-oriented processing is not up for debate; developers would be thrilled by any new capabilities beyond blinking LEDs and custom debugging tools. Rather, the key issue is determining the viability of providing such capabilities, which have often eluded much more powerful networks and systems.

Drawing from the capabilities and experiences of a broad spectrum of previous work, we have designed Wringer, a rapid-prototyping framework that allows for triggered custom processing, conditioned on the failure (or success) of any predicate defined over local or distributed network state. However, we are quick to note that this tool is not a one-stop solution; there are a host of tasks for which it is ill-suited or even incapable. Rather, in such a complex environment, a host of tools are needed to cover the complete spectrum of user needs, many of them complementary. Perhaps it is too soon to begin discussing tool interoperability, but as demonstrated by our system's ability to utilize the capabilities of specialized instrumention frameworks, this is a beneficial direction to take this field of research in.

7 References

[1] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Providing an interactive environment for wireless debugging and development," *IPSN*, 2006.

[2] J. Yang, M. L. Soffa, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," *SenSys*, 2007.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," *USENIX*, 2004.

[4] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, "X-trace: A pervasive network tracing framework," *Networked Systems Design and Implementation*, 2007.

[5] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse, "Ants: A toolkit for building and dynamically deploying network protocols," *IEEE OpenArch*, 1998.

[6] P. Levis and D. Culler, "Mate: A tiny virtual machine for sensor networks," *ASPLOS*, 2002.

[7] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler, "Design and implementation of a sensor network system for vehicle tracking and autonomous interception," *IEEE EWSN*, 2005.

[8] "The revised(5) report on the algorithmic language scheme." http://swiss.csail.mit.edu/projects/scheme.

[9] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica, "The design and implementation of a declarative sensor network system," *UC Berkeley EECS Department Technical Report*, 2006.

[10] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," *SenSys*, 2005.

[11] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis, "Visibility: A new metric for protocol design," *SenSys*, 2007.

[12] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," *USENIX*, 2006.

[13] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," *NSDI*, 2007.

[14] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks wtih envirlog," *INFOCOM*, 2006.

[15] D. Gunter, B. L. Tierney, C. E. Tull, and V. Virmani, "On-demand grid application tuning and debugging with the netlogger activation service," *GRID*, 2003.

[16] A. I. Tomlinson and V. K. Garg, "Detecting relational global predicates in distributed systems," *ACM/ONR workshop on Parallel and Distributed Debugging*, 1993.

[17] R. Mehmood, J. Crowcroft, S. Hand, and S. Smith, "Grid-level computing needs pervasive debugging," *Grid Computing Workshop*, 2005.