PerViz: Painkillers for Pervasive Application Debugging

Hubert Pham and Justin Mazzola Paluska
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, U.S.A.

Abstract—**Debugging pervasive applications is difficult due to their distributed, asynchronous, and dynamic nature. To help ease the debugging process, we propose PerViz, a developer-targeted tool that enhances system visibility through real-time visualizations of system state, semi-automates application restarts and positioning, and enables both real-time and asynchronous collaboration in debugging between developers. Developers interact with PerViz through a web browser, which provides a convenient, centralized location to study and filter aggregated application debugging logs and state. In our experience with using PerViz, we have found its log aggregation and real-time visualizations to be key facilitators for effective debugging.**

I. Introduction

Debugging pervasive computing applications is a challenging affair. Several factors conspire to its difficulty, such as the distributed and asynchronous nature of these applications. For instance, when something goes wrong, the developer must chase down debugging output from multiple components, scattered across multiple processes and hosts, and then piece together reams of separate logs in order to form a mental picture of the system state. Still, the logs may be deliberately terse or non-existent, because the detailed logs, with their typical minutiae and detritus, had previously overwhelmed the developer.

It might then take several repeated restarts of the application before the problem is apparent. At best, the developer may only need to adjust the appropriate level of debugging output, but worse, the problem may not be easily reproducible due to non-deterministic timing bugs. Each restart incurs cognitive overhead in repositioning the distributed system back to its state before failure—the develop-run-test loop is slow and grows slower with the addition of each module.

Finally, once all the relevant logs are collected, the developer must sift through and map low level messages to higher-abstraction system state and events—generally a very manual process.

While these issues plague distributed applications in general, the debugging process is further exacerbated in pervasive applications, which tend to also be dynamic, distributed, and heterogeneous.

A. Something Must Be Done

We have faced all of these challenges throughout our experience building pervasive applications [1]. Our frustrations prompted us to investigate approaches and tools to facilitate pervasive application debugging.

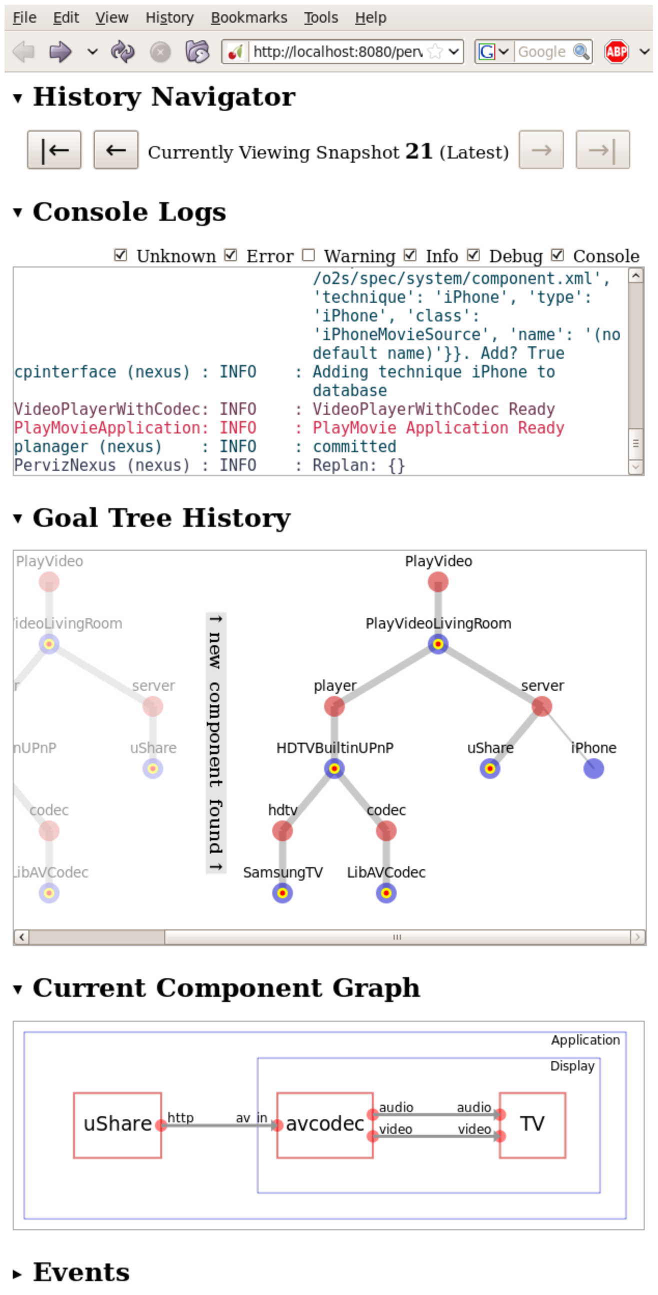
The central hypothesis that this paper explores is that effective debugging demands techniques that enhance system visibility, enable rapid reproducibility, and support collaborative debugging among developers:

- **Visibility** Because system state and configuration is distributed, it is essential to centralize all information in a single, localized environment. Effective user interfaces should then visualize and enable inspection of system components—both during runtime and in post-analysis.
- **Reproducibility** Reproducing a scenario should be simple and tool-assisted. To help position the system to a known desired state, tools should enable developers to selectively replay previously logged—or newly contrived—system inputs and events to a running system.
- **Collaborative Debugging** Enabling collaborative debugging among developers, either concurrently or asynchronously, can potentially leverage divide-and-conquer effectively, while bringing individual specialties to bear on different subsystems.

B. Contributions

This paper presents PerViz, a tool to facilitate debugging pervasive applications through interactive, real-time visualization (Figure 1). Unlike existing projects that typically focus either on high-fidelity capture of low-level application state and timing, or effective visualizations of code execution and algorithms, PerViz instead concentrates on enhancing application visibility and control at the middleware- and platform-layer. While PerViz does not introduce any fundamentally new techniques, it integrates the features below into a unified tool, an approach that we believe is unique. These features include:

- **Log Aggregation and Filtering** PerViz captures all debugging output from all relevant processes and delivers them to a central location where they are combined for viewing or interactive filtering.
- **Real-time Visualization** PerViz generates and displays real-time visualizations of system state. In our implementation, such state includes decision trees and diagrams that depict the running component graph. These visualizations are interactive and enable the developer to drill-down and inspect the state of running components. PerViz also depicts network flows and system



```
                               /o2s/spec/system/component.xml',
                               'technique': 'iPhone', 'type':
                               'iPhone', 'class':
                               'iPhoneMovieSource', 'name': '(no
                               default name)'}}. Add? True
cpinterface (nexus) : INFO    : Adding technique iPhone to
                               database
VideoPlayerWithCodec: INFO    : VideoPlayerWithCodec Ready
PlayMovieApplication: INFO    : PlayMovie Application Ready
planager (nexus)    : INFO    : committed
PervizNexus (nexus) : INFO    : Replan: {}
```

Figure 1. PerViz provides a web-based interface to inspect running systems in real-time. Here, the system just found a new potential component for use.

events on the component graph to help developers quickly deduce the source of problems.

- **Time Machine and Assisted Replay** PerViz also enables developers to review and study the system offline from stored logs. A VCR-like interface provides controls for stepping through the application's run to recall stored state. Furthermore, because PerViz records all system events, developers can quickly reposition a restarted application by re-sending previously captured events or inputs (or fabricating new ones).
- **Salience Detection** PerViz can depict salient, high-level events (such as application adaptation) by leveraging existing planning processes that detect and react to runtime inputs. For example, component failures might prompt an application's planning process to change the system to use alternative resources. PerViz exploits this knowledge to delineate that an important system change has occurred, as well as the reason for that change.
- **Web Browser as the User Interface** The developer's user interface is completely web-based, an approach which offers several advantages such as ease of interactive synchronous collaboration, where multiple developers view the same, dynamic system state on their own screens; simplified asynchronous collaboration, where developers can share URLs that identify and recall a particular application state; and minimal requirements to run, needing only a capable browser, thus transforming even hand-held devices into mobile debugging consoles. This last feature is particularly useful when developers need to physically investigate systems that are far from their workstations.

To test our approach, PerViz targets applications built using our Planner [2] and Components platform [3]. However, the principles are general, and PerViz features APIs to enable integration with other systems.

We have used PerViz to debug our applications and found that runtime log aggregation and filtering is a primary facilitator for effective debugging. Second in usefulness are the visualizations of system-calculated decisions and state, since they typically enable us to quickly pinpoint the problem source. Finally, the web-centric approach provides convenience through its centralized platform for investigation, system control, and collaboration, all while affording physical mobility.

C. *Paper Outline*

In this paper, we discuss the approach and implementation of PerViz. Section II discusses system architecture and implementation. Section IV discusses limitations and applicability of PerViz, while Sections III and V evaluate the tool: Section III describes our experience using the tool with case studies, and Section V presents microbenchmarks. Section VI reviews related work, and Section VII concludes.

II. System Architecture

A tool like PerViz faces two main challenges. The first is providing system information in a timely manner while imposing minimal performance penalties and side effects. The second challenge concerns the retrieval, organization, archiving, and presentation of distributed system information in a manner that is both useful and effective for application debugging. This section begins with an overview of PerViz's

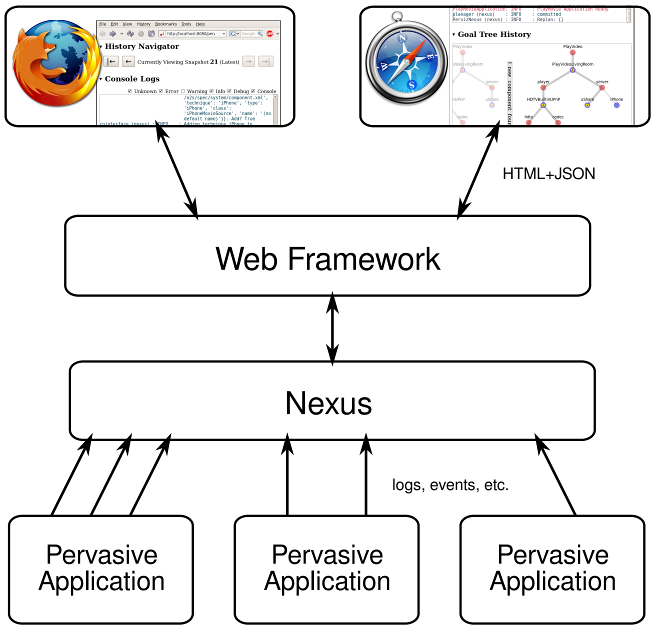


Figure 2. The PerViz Nexus aggregates logs and events from instrumented applications and forwards that data for presentation in the browser.

architecture and follows with a detailed discussion of how PerViz meets the challenges above.

A. Overview

Figure 2 depicts PerViz's overall architecture. The instrumented application is represented as the bottom layer. PerViz comprises the upper layers, in addition to generic tools that aid with instrumentation. The state Nexus, the central process in PerViz, collects and organizes an extensible set of messages from the application, such as debugging output, application decisions, system events, and information about network flows. The web framework implements an HTTP interface that exposes the collected information and serves the user interface (implemented as HTML and JavaScript [4]) to potentially multiple web browsers. Developers interact with PerViz through their web browser to inspect their running applications.

Broadly speaking, PerViz itself is a generic tool for aggregating, archiving, and presenting messages in real-time, all through well-specified interfaces. However, to derive maximal utility from PerViz, the underlying application (or the middleware on which the application runs) must be instrumented with platform-specific code to collect, marshal, and forward platform-specific system state to the Nexus. In addition, the in-browser client-side library must be augmented to interpret those new states and generate appropriate visualizations.

B. Application Middleware

To experiment with PerViz, we used our Goal-oriented application Planner [2] combined with our Components system [3] as the application middleware. Our middleware targets pervasive applications that naturally decompose into two parts: a planning process (or context manager) responsible for making high-level decisions (e.g., adaptation, resource allocation, or contextual decisions) during the life of the application, and a component-based implementation, on which the application executes. The Goal-oriented Planner facilitates the run-time planning process, while the Components platform provides an adaptive, distributed object environment.

While this section focuses on instrumentation approaches in the context of our middleware platform, many of the techniques presented have general applications, and we are confident that they can be applied to similar platforms like BASE [5], PCOM [6], and one.world [7].

1) Overview: The Goal-oriented Planner provides an extensible mechanism for managing system run-time decisions governed by user requests. In response to user requests, the Planner asserts or revokes high-level Goals (e.g., "Play Video") and searches the environment for Techniques (e.g., "Play Video using Nearby Living Room Devices") that best satisfy the user's Goals. Techniques, available from an open-ended variety of sources, codify recipes for satisfying specific Goals. The middle of Figure 1 shows a visualization of a typical "Goal Tree" that the Planner generates internally to satisfy an asserted Goal. At the top of the tree is a node representing the asserted Goal, with descendant nodes representing available Techniques, any of which might satisfy the Goal. A Technique may have sub-Goals as prerequisites, all of which must be satisfied in order for that Technique to succeed. Ultimately, the chosen Techniques generate snippets of code that target our Components platform for execution.

The Components platform provides middleware for assembling pervasive applications that demand fault-tolerance and adaptivity in distributed, dynamic environments. Applications are written as single-threaded programs that use a simple, block-diagram construction API to instantiate, configure, assemble and monitor a set of distributed components. Components are composable and feature named *connectors*, which serve as input/output ports for high-speed communication links between modules (Figure 1, bottom).

The application, also known as constructor logic, monitors the autonomous component network via a stream of high-level events that indicate new resource availability or failures. Consequently, the constructor logic provides a convenient, centralized conduit for controlling reconfiguration through component hotswapping.

2) Planner-Component Integration: Techniques house recipes for satisfying user-requested Goals; the recipes typically employ the Components' construction API to find and create a component graph at run-time that implements desired functionality. In addition, Techniques monitor the operation of the running component network through a serial stream of high-level events originating from the network. Changes in the environment, such as failures in the component graph, may cause running Techniques to fail, thus prompting the Planner to adapt the application by searching for alternative Techniques. Similarly, new resources that become available at run-time enable the relevant Techniques to become viable, prompting the Planner to upgrade the application.

C. Application Instrumentation

A common instrumentation requirement, regardless of platform, involves capturing all debugging output (logging) from relevant processes in real-time. PerViz provides a platform-agnostic mechanism to capture logging for delivery to the Nexus. The next two requirements are semi-specific to our choice of platform but are generally applicable: 1) capturing frequent, low-level messages, and 2) detecting high-level, salient events. In the context of our platform, low-level messages include the serial stream of events between the component network and constructor logic, as well as statistics on data flow between the components themselves. High-level salient occurrences include adaptation decisions, along with the circumstances that prompted them. The challenge lies in capturing all of this information in a way that imposes minimal disruption to the running application while without compromising timeliness.

1) Capturing Logs: To capture debugging output from all component processes, PerViz provides a generic, lightweight harness program, implemented in Python, which invokes a specified target process (e.g., a single component). The harness captures all output from the target's standard output and error using an efficient `select` loop. Captured output is timestamped and sent to the Nexus, where it is combined with logging output from other harness processes, for presentation. We rely on an out-of-band mechanism (such as NTP [8]) to synchronize time across processes for developer convenience, although time synchronization is not strictly necessary (as further discussed in Section IV).

Running the harness in a separate process than that of its target offers several advantages. First, the harness remains fairly generic. Second, there is a lesser chance that capturing and delivering debugging output will greatly interfere with the functioning of the target process due to network latency or blocking. By relying on the efficiency of the operating system scheduler, the harness can reliably consume the logging output of its target to prevent the target process's buffer from overflowing, while independently forwarding to the Nexus.

2) Capturing Events and Connector Statistics: We instrument the Components platform to send an additional copy of all events to the Nexus, as events are fairly lightweight data structures. Events are self-describing, and contain unique identifiers of the source (emitting the event) and the destination, enabling the Nexus to later replay those events to the correct recipient. Similarly to the harness, the Components platform sends cloned events to the Nexus in a separate thread to minimize disruption.

Connectors, high-speed data communication ports between components, are only lightly instrumented. Because the amount of data that flows on connectors is typically large (e.g., media streams), we do not send a copy of that data to the Nexus. Instead, connectors simply indicate to the Nexus the mere fact that data has been sent (and how much).

In practice, we have found the approach more than sufficient for debugging: often it is more useful to know whether data is flowing rather than the actual binary contents of that data. Since we do not automatically capture the actual data, it is not possible to replay connector data between components—although it is possible to inject newly fabricated data through a connector. We have found the limitation to be rarely a problem; replaying events that ultimately trigger data flow across connectors has been sufficient for system repositioning.

3) High-level Decisions: The Nexus detects changes in component availability via heartbeat keep-alive connections built into the Components middleware platform. While health changes are reported in the user interface, it is often more useful to convey whether they alter high-level decisions by the Planner.

Changes in component availability affect the viability of relevant Techniques, prompting the Planner to re-evaluate the current application implementation decisions in its Goal Tree. We instrument the Planner in two ways: first, the logic that prompts the planner to re-evaluate its Goal Tree now also determines and publishes a reason for re-evaluation (e.g., certain, specific components just became available), and second, the (new) tree is sent to the Nexus for presentation. To the client-side user interface, the arrival of a new Goal Tree, along with the reason that prompted it, signifies an important change to the application state.

4) State That Is Not (Automatically) Captured: While our adaptation of PerViz captures and archives events, the timing of connector flows, and high level decisions, it does not capture the internal state of individual components. It is possible to inspect these components from the user interface during run-time, but doing so only reveals the *current* (not past) state of the component.

D. Nexus and Web Framework

The centralized Nexus maintains a history of collected application messages, e.g., real-time debugging logs and events. The Nexus is application-generic and treats all incoming messages, encoded in JSON [9], as opaque. There are several ways to deliver messages to the Nexus: the Nexus exports a standard Python API, so the application instrumentation can deliver messages in arbitrary ways to an adapter which runs in-process with the Nexus. Alternatively, the instrumentation can directly POST messages to the Nexus via a simple HTTP interface.

The Nexus packages incoming messages in a lightweight "snapshot" container, assigns the snapshot an ID (from a set of monotonically increasing integers), and appends it to a log of snapshots. Hence, developers can study state snapshots that depict the evolution of a running application, using VCR-like operations such as pausing, rewinding, and fast forwarding.

The Nexus also runs a web server, which exposes the snapshots via a simple, RESTful [10] HTTP API (e.g., `GET /state/snapshot/12`). The web server serves HTML and JavaScript files that retrieve snapshots from the Nexus and visualize application state in the browser.

To manipulate running applications (e.g., by injecting recorded system events for repositioning), the Nexus must be augmented with a platform-specific adapter to handle such operations. So at the developer's request, the browser sends an HTTP POST to the Nexus, which in turn passes control to the adapter for handling.

1) Server–Browser Communication: The user interface is completely browser-based, which brings certain advantages, such as enabling collaborative debugging between developers (by affording different views of the same system, simultaneously, across multiple browsers) and providing a convenient environment to dereference URLs that refer to a particular point in an application's run, past or present (thereby enabling bookmarking and sharing).

The PerViz user interface consists solely of HTML and JavaScript, which makes heavy use of the now common AJAX pattern to communicate with the server without needing to reload the page. PerViz communicates with the server to either retrieve new snapshots or to request certain actions (e.g., to replay an existing event or to invoke a Goal for satisfaction).

However, using the browser presents certain challenges, especially in timeliness. In order to mitigate latency when requesting snapshots, rather than poll the server at regular intervals, PerViz instead uses COMET [11], a technique in which the server blocks requests until there is relevant data to return. Specifically, requesting a non-existent snapshot ID (because it has not yet been assigned to a message) causes the server to accept the connection but not return any data until the Nexus assigns a new message to that ID.

2) Real-Time Visualizations: Platform-specific visualizations of the Goal Tree and component graph are generated completely client-side using JavaScript and HTML5 Canvas [12]. The tree layout algorithm employs standard techniques that walk the tree and compute hints for node positioning. The layout algorithm for drawing component graphs first computes a partial ordering of all components based on their data flow connections, followed by a topological sort to compute a total ordering for individual component positioning.

To visualize network flows, client-side data structures store coordinates of all elements representing connectors. Snapshots that indicate data flow contain connector IDs, enabling PerViz to highlight the visual element representing the relevant connection.

E. Extensibility

While PerViz currently targets our Planner and Component middleware platform, its coupling to the middleware exists mainly in instrumentation. We are confident that PerViz can be adapted with modest effort to other frameworks. PerViz's harness for capturing debugging output, the Nexus and its data structures for storing snapshot history, and the HTTP interface for browser communication is general and not specific to our middleware. To adapt PerViz for use with other platforms, one would need to determine the application state worth visualizing, implement the necessary instrumentation to marshal and deliver that state to the Nexus, and write the JavaScript plug-in(s) to visualize it.

Because the Nexus imposes relatively few restrictions on the data that it collects and forwards to the web browser, there are a few alternative visualization approaches possible with PerViz:

1) Direct Component Visualization: Instead of writing JavaScript plug-ins to interpret platform-specific messages, another approach is to visualize application state directly in the relevant application process. Developers can augment application modules with a lightweight web-server, which serves pages that display arbitrary visualizations in real-time. PerViz can seamlessly incorporate such pages into the main user interface by use of HTML `iframes`.

2) Salience Triggers: Our platform adaptation of PerViz relies on the middleware to signal significant and salient changes in application state. However, it is often useful during debugging to have certain conditions (e.g., breakpoints) halt execution or be specially logged. Developers can implement these features by augmenting the instrumentation to trigger on desired conditions and writing simple JavaScript plugins to map those events into meaningful representations.

III. Case Studies

In this section, we distill our experiences with using PerViz. The application we examine is JustPlay [1], a home automation application that relieves the user of the task of configuring (and re-configuring) home electronics. JustPlay uses the Planner and Components platform to demonstrate a practical application that manages, configures, and controls audio and video devices in a typical living room environment. In a typical scenario, a user may speak an underspecified goal like "Play the Simpsons Movie". The system responds by searching for goal-oriented ways to comply, given resources in the environment.

We have found that log aggregation with filtering is key to efficiently pinpoint problems. In addition, seeing what decisions the system makes, along with the existence of network flows in real-time, improves the debugging experience.

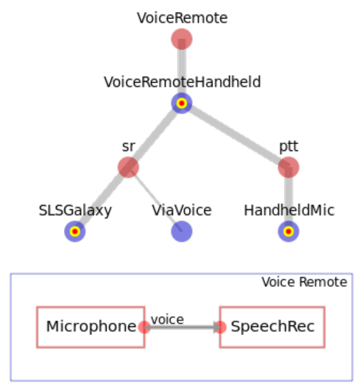


Figure 3. The VoiceRemote Goal Tree (top) and resulting implementation (bottom). Nodes in bullseye represent chosen Techniques.

A. *Voice Remote*

In JustPlay, users communicate with the system principally by speech. They speak into a microphone on a handheld device, which captures audio and delivers the speech to a speech recognizer component, running on a different process. Figure 3 illustrates the resulting Goal Tree and component network that implements the voice remote. The speech recognizer packages recognized text into an event that is ultimately delivered to the Planner. The Planner reacts to the event by invoking another Goal or modifying already running Goals.

Quite often, uttering a command into the microphone results in no perceptible response. A number of things might be wrong: perhaps in setting up the voice remote application, the Planner selected the wrong microphone. Perhaps the audio capture process on the handheld device crashed. Or maybe the audio capture is fine, but the network connection between the handheld and the speech recognizer is disconnected. The problem could lie in the speech recognizer: either it crashed, or maybe it couldn't recognize the speech. Or the speech recognizer works, but the Planner is misinterpreting the event.

We have experienced all of these faults. Our debugging technique prior to PerViz would have been to find each component by logging into the appropriate hosts and scanning through logs. If the debugging output was too sparse, we would have needed to add more `printf` statements—potentially in each component—before re-deploying.

Using PerViz to investigate this scenario, we can immediately verify that the voice remote is properly setup from a cursory glance at the Goal-Tree and component visualization (Figure 3). We can also further inspect the Goal Tree and component graph by clicking on their visual elements, to ensure that the desired resources are selected. Uttering a command into the handheld microphone, we see, on the component graph, the network connection between the handheld and speech recognizer light up. However, we notice that the speech recognizer fails to send an event to the Planner. We thus filter out all debugging output except for that of the speech recognizer, but we find nothing apparent. Nonetheless, PerViz has helped us pinpoint the likely culprit, and upon further investigation (i.e., by logging into the recognizer's host), we discover that the recognizer had hung.

B. *Streaming Media*

After repairing the speech recognizer, we are then able to utter "Play the Simpsons Movie" and see that an event fires from the recognizer to the Planner. The Planner subsequently generates a new Goal Tree and component-based implementation. The implementation consists of a movie source component that streams MPEG2 movies over HTTP to a display component, which itself consists of a Python wrapper around VLC [13], an open-source software video player.

That said, we witness no video playback, but we immediately see why: an uncaught exception in the Python VLC wrapper, due to a failed assertion. The browser displays the exception stack trace prominently within the log aggregation (Figure 4). In our experience, a large class of errors is quickly caught simply by having debugging output and error tracebacks aggregated and filtered in one place.

IV. Applicability and Limitations

We now turn to a discussion on the applicability and limitations of a tool structured like PerViz.

A. *Reasoning About Causality*

PerViz aims to provide developers with enough tools to assist them in comprehending application behavior. One important behavior is *causality*, or what events cause other events to occur. PerViz does not automatically capture causality in a formal sense, but rather gives programmers tools to easily piece together causal relationships as needed.

For instance, through the use of a central process that collects emitted messages from distributed components, PerViz guarantees that all messages originating from an individual component arrive and are displayed in the same order emitted. Modulo multiple threads producing debugging output within an individual process, logs for that process are causally ordered (with respect to other output from the same process) at the central PerViz console. Given that a common debugging technique is the use of `printf`'s, so long as the output arrives in a consistent order, the programmer can deduce problems just as they would with traditional single-process programs. PerViz's contribution is ensuring accurate and timely delivery of debugging output from distributed components to a central location, combined with affordances for filtering.

To address causality across distributed components (rather than simply within a single module), PerViz captures certain communication and messages between components. For applications that rely on message passing or data streams among modules, PerViz can capture and visualize the sending and receiving of messages. A developer looking at `printf`'s in one process can see that it creates a message, use PerViz to track where the message goes, and then continue reading `printf`'s in the destination process.

We have found that even though PerViz does not automatically capture causality, this "limitation" does not materially diminish the utility of PerViz. First, PerViz need not depend on high resolution time synchronization among components, thus broadening its applicability. Second, through its simple guarantees and affordances, PerViz remains easy to understand while still able to assist developers when they need to piece together causality.

B. *Handling Common Faults*

Our PerViz implemention handles common application faults, such as process crashes, host crashes, and severed network connections. PerViz, by itself, addresses some of these issues, but combined with our particular middleware platform, PerViz handles all of them.

As mentioned, to capture an application process's debugging output, PerViz provides a harness that runs alongside the application but in a separate OS process, capturing the application's standard output and error. As such, the harness imposes minimal overhead on the application (as compared to running the harness inside the application's process) and is resilient to application crashes.

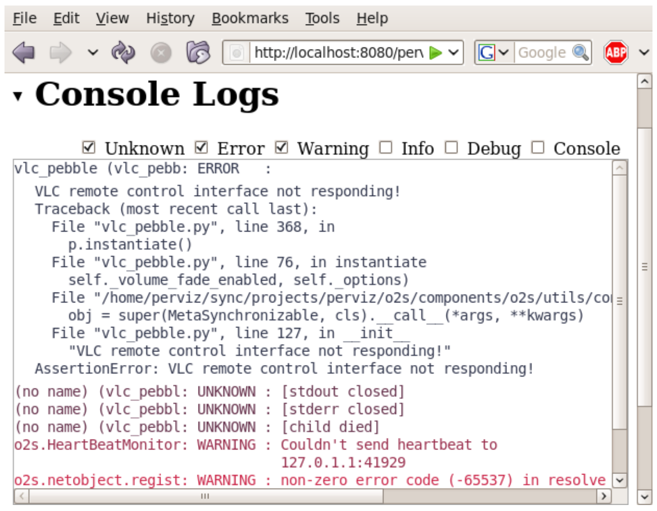
To address host crashes and severed network connections, our middleware provides a heartbeat mechanism that monitors component liveness. A more general approach might involve maintaining a TCP connection with keep-alive between the harness (on the application's host) and the Nexus.

C. *PerViz in a Production Environment*

PerViz targets lab environments, aiding developers with prototyping and debugging. PerViz, as implemented, is not yet focused towards easing debugging applications deployed in the field. However, future work might address this limitation by exploring how tools could provide visibility in running, production systems. One candidate approach involves components recording debugging output in an internal circular buffer to provide both current and recent log messages. A front-end like PerViz could then plug into a running application, extract, and display recent state and events to aid troubleshooting. A challenge for such an approach is to appropriately match the logging rate and buffer size across distributed components so that their respective histories roughly correspond.

V. MICROBENCHMARKS

To show that application log messages appear in a timely manner, we measure the throughput and latency of capturing


File Edit View History Bookmarks Tools Help
http://localhost:8080/perv
Google
Console Logs
☑ Unknown ☑ Error ☑ Warning ☐ Info ☐ Debug ☐ Console

```
vlc_pebble (vlc_pebb: ERROR   :
  VLC remote control interface not responding!
  Traceback (most recent call last):
    File "vlc_pebble.py", line 368, in
      p.instantiate()
    File "vlc_pebble.py", line 76, in instantiate
      self._volume_fade_enabled, self._options)
    File "/home/perviz/sync/projects/perviz/o2s/components/o2s/utils/co
      obj = super(MetaSynchronizable, cls).__call__(*args, **kwargs)
    File "vlc_pebble.py", line 127, in __init__
      "VLC remote control interface not responding!"
  AssertionError: VLC remote control interface not responding!
(no name) (vlc_pebbl: UNKNOWN : [stdout closed]
(no name) (vlc_pebbl: UNKNOWN : [stderr closed]
(no name) (vlc_pebbl: UNKNOWN : [child died]
o2s.HeartBeatMonitor: WARNING : Couldn't send heartbeat to
                                127.0.1.1:41929
o2s.netobject.regist: WARNING : non-zero error code (-65537) in resolve
```

Figure 4. Filtering controls help remove low-level debugging so that errors become more apparent. Here, the aggregated debugging console distinctly shows an error traceback within a remote application component. Debugging messages are color-coded to their respective sources.

and delivering such messages from the application to the browser. All components are local for these measurements. Figure 5 plots the end-to-end time between when a message is emitted to the time it appears in the browser (via the harness and Nexus). The typical latency is around 30 ms, but we find even with the worst case-latency of 80 ms imperceptible.

Figure 6 suggests a sustained throughput near 60 messages per second. While we can further improve throughput by batching incoming messages within a small time window at the Nexus, we have found the current rate satisfactory for our applications.

VI. RELATED WORK

Related projects generally fall into one of two categories: instrumentation techniques or system visualization approaches. Unlike similar projects, PerViz focuses on bridging instrumentation and interactive visualization, at the application middleware level.

A. *Instrumentation and Debuggers*

There have been several approaches proposed for instrumenting and debugging distributed applications. One is to run distributed processes in a virtualized environment, proposed by projects like PDB [14], [15], which focus on grid computing applications. By running application processes on virtual machines, developers have greater visibility and finer control over the distributed application's execution. However, there are practical limits to the approach, as the (distributed) application under test typically must run completely in a single virtualized environment, on a single host. While such debuggers offer low-level control and visibility (e.g., at CPU instruction and memory-access level), they are

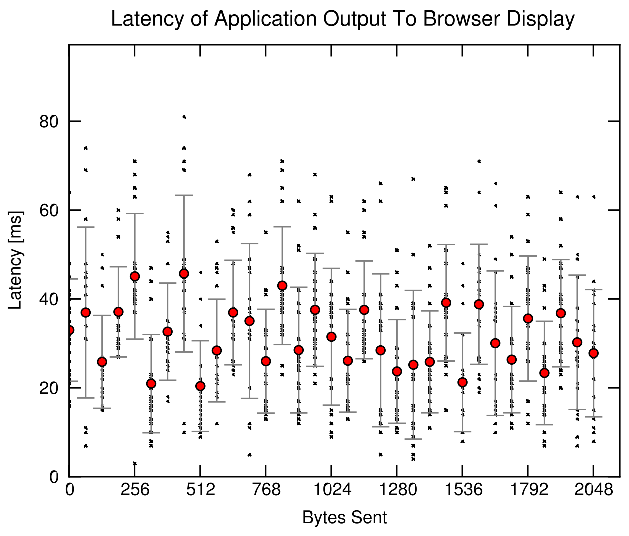


Figure 5. Average latency, in red, remains relatively consistent around 30 ms. Each circle represents the mean of 20 samples (shown in black).

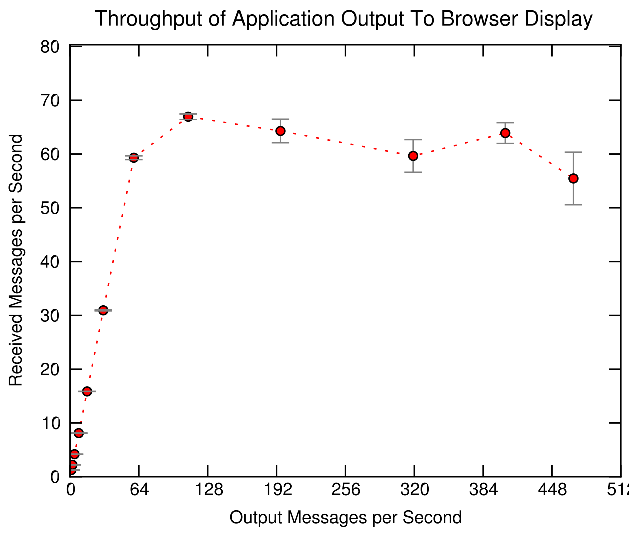


Figure 6. Sustained throughput tapers at 60 messages/second, or a message roughly every 16ms. Each point represents the average of 10 runs.

less appropriate for tracking higher-level events (e.g., at the middleware or platform level).

Even so, PerViz is partly inspired by projects like SimOS [16], which pioneered virtualization-based techniques for 1) repositioning traditional, single-process applications (or operating systems) to a desired state, and 2) capturing high-rate data efficiently. SimOS employs various modes to vary the fidelity of application simulation, and provides hooks for developers to annotate, or map, generated system events into higher-level knowledge. PerViz borrows these concepts in targeting distributed pervasive applications.

In contrast to SimOS, `liblog` [17] proposes a completely user-space approach to logging distributed application events for both further study and replay. `liblog` captures application level calls to `libc` and focuses on achieving very high-fidelity capture and deterministic replay rather than providing high-level visibility and interaction.

For distributed applications that exhibit extensive data flow between components, Nigam [18] proposes the use of transmission rates as a metric for real-time health monitoring. We apply this concept in instrumenting the data connections between components. Future work might include applying advanced data flow measurement techniques proposed in Remos [19].

B. Visualization

The Event Heap Visualizer [20] depicts the iRoom [21] tuple-space by encoding state into an abstract visualization of shapes, colors, positions, transparency values, and sizes. Like the USS monitor [22] project, it serves as an ambient display to imply application structure and activity through trends, rather than as an interactive tool for direct system investigation.

On the other hand, software visualization techniques, such as Baecker's work [23] and broadly surveyed by Bassil and Keller [24], aims to enhance program comprehension by visualizing low-level algorithms, code execution, and state with animations. While traditionally time-consuming to create, projects like Lens [25] provide tools to enable developers to rapidly build these animations. We view these projects as orthogonal to PerViz, providing deeper and lower-level analysis for individual components.

VII. Conclusion

Despite having been proposed for about 20 years, pervasive applications are still confined to research labs and have enjoyed relatively limited commercial adoption. We believe that one reason is that they are still too challenging to build and debug to gain traction among mainstream developers. To improve pervasive computing's appeal and foster its widespread application, we propose a greater investment in engineering effort towards enhancing ease of development.

This paper presents PerViz, which takes a step towards easing development by reducing several debugging pain points. PerViz provides real-time application visibility through log aggregation and state visualization; enhances developer efficiency by assisting with application repositioning; and enables collaboration among developers through the use of a web-based interface. It bridges instrumentation and interactive visualization, offering generic infrastructure for aggregating, archiving, and presenting system state and run-time logs of distributed components. With its plug-in architecture, PerViz is designed to work with arbitrary applications, after a modest investment in instrumentation.

In our initial experiments, we have found PerViz to be effective in debugging applications, with the principal benefits arising from real-time visualizations of the application and the aggregation of logs to a central interface.

VIII. Acknowledgments

This work is sponsored by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan. We would like to thank Steve Ward, Larry Rudolph, Robert C. Miller, and our shepherd, Xun Luo, for their helpful comments.

References

[1] J. Mazzola Paluska, H. Pham, U. Saif, C. Terman, and S. Ward, "Reducing configuration overhead with goal-oriented programming," in *PerCom*, 2006.

[2] J. Mazzola Paluska, H. Pham, U. Saif, G. Chau, C. Terman, and S. Ward, "Structured decomposition of adaptive applications," in *PerCom*, 2008.

[3] H. Pham, J. Mazzola Paluska, U. Saif, C. Stawarz, C. Terman, and S. Ward, "A dynamic platform for runtime adaptation," in *PerCom*, 2009.

[4] "ECMAScript language specification," Ecma International, Tech. Rep. ECMA-262, 1999.

[5] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel, "BASE: a Micro-Broker-Based middleware for pervasive computing," in *PerCom*, 2003.

[6] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM - a component system for pervasive computing," in *PerCom*, 2004.

[7] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, S. Gribble, T. Anderson, B. Bershad, G. Borriello, and D. Wetherall, "Programming for pervasive computing environments," *University of Washington Technical Report UW-CSE-01-06-01*, 2001.

[8] D. Mills, "Network time protocol, version 3," IETF, RFC 1305, 1992.

[9] D. Crockford, "JavaScript object notation," IETF, RFC 4627, 2006.

[10] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, UC Irvine, 2000.

[11] D. Crane and P. McCarthy, *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* Apress, 2008.

[12] I. Hickson, "HTML5 standard," WHATWG, Working Draft, 2009.

[13] "VLC media player," http://www.videolan.org/vlc/.

[14] A. Ho, S. Hand, and T. Harris, "PDB: pervasive debugging with xen," in *IEEE/ACM GRID*, 2004.

[15] T. L. Harris, "Dependable software needs pervasive debugging," in *ACM SIGOPS*, 2002.

[16] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the SimOS machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 1, pp. 78–103, 1997.

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

[18] A. Nigam, "Analytical techniques for debugging pervasive computing environments," M. Eng. Thesis, MIT, 2004.

[19] P. Dinda, "Design, implementation, and performance of an extensible toolkit for resource prediction in distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 2, pp. 160–173, 2006.

[20] M. R. Morris, "Visualization for casual debugging and system awareness in a ubiquitous computing," in *Ubicomp*, 2004.

[21] B. Johanson, A. Fox, and T. Winograd, "The interactive workspaces project: Experiences with ubiquitous computing rooms," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 67–74, 2002.

[22] K. Kang, J. Song, J. Kim, H. Park, and W. Cho, "USS monitor: A monitoring system for collaborative ubiquitous computing environment," *IEEE Transactions on Consumer Electronics*, vol. 53, no. 3, pp. 911–916, 2007.

[23] R. Baecker, C. DiGiano, and A. Marcus, "Software visualization for debugging," *Commun. ACM*, vol. 40, no. 4, pp. 44–54, 1997.

[24] S. Bassil and R. Keller, "Software visualization tools: Survey and analysis," in *IWPC*, 2001.

[25] S. Mukherjea and J. T. Stasko, "Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger," *ACM TOCHI*, vol. 1, no. 3, pp. 215–244, 1994.