Debugging Cyber-Physical Systems with Pharo

An Experience Report

Matteo Marra
Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
{mmarra, egonzale}@vub.be

Steven Costiou
Mickaël Kerboeuf
Alain Plantec
Lab-STICC UMR CNRS 6285
University of Western Brittany
{steven.costiou, mickael.kerboeuf, alain.plantec}@univ-brest.fr

Guillermo Polito
Stephane Ducasse
RMoD - Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
{guillermo.polito, stephane.ducasse}@inria.fr

Abstract

Cyber-Physical Systems (CPS) integrate sensors and actuators to collect data and control entities in the physical world. Debugging CPS systems is hard due to the time-sensitive nature of a distributed applications combined with the lack of control on the surrounding physical environment. This makes bugs in CPS systems hard to reproduce and thus to fix. In this context, on-line debugging techniques are helpful because the debugger is connected to the device when an exception or crash occurs.

This paper reports on our experiences on applying two different on-line debugging techniques for a CPS system: remote debugging using the Pharo remote debugger and our IDRA debugger. In contrast to traditional remote debugging, IDRA allows to on-line debug an application *locally* in another client machine by reproducing the runtime context where the bug manifested. Our qualitative evaluation shows that IDRA provides almost the same interaction capabilities than Pharo's remote debugger and is less intrusive when performing hot-modifications. Our benchmarks also show that IDRA is significantly faster than the Pharo remote debugger, although it increases the amount of data transferred over the network.

Keywords Cyber-Physical Systems, software tools, debugging

1. Introduction

In the recent years we have witnessed a revolution in networking technology and mobile computing making embedded computers get every day smaller and more powerful. This in turn has boosted the development of a novel type of distributed systems called Cyber-Physical Systems (CPS) which integrate sensors and actuators to collect data and control entities in the physical world. Broadly speaking, CPS systems are distributed applications that track, observe and analyse large collections of data from computerized entities, e.g. robots on a factory floor, vehicles and sensor-equipped road infrastructure, etc. Examples of CPS systems are diverse and include smart grids, medical care systems, autonomous vehicle systems, robotics systems integrated in smart factories, etc.

CPS systems exhibit a number of characteristics which distinguishes them from traditional distributed embedded systems. Like embedded systems, CPS systems consist of a number of interconnected devices with limited resource constraints. However, the main goal of a CPS system is to remain responsive to environment changes and network commands. As such, the massive amount of data being collected by sensors requires the execution to have safe boundaries of time while delivering consistent and accurate operations. In addition, time synchronization has to take into account the inherent latency of the networking layer between the physical entity and the algorithms monitoring and running analysis on the received data.

In this paper, we focus on debugging support for CPS systems (cf. Section 2). Debugging distributed systems is hard because developers must deal with the inherent non-determinism of concurrent processes and partial failures. This complicates the debugging task since an error detected in one execution might not manifest itself in the debugging session. Furthermore, the mere presence of the debugger

might exacerbate this non-determinism by affecting the way in which the program behaves [MH89]. Debugging CPS systems is *even* harder due to the time-sensitive nature of CPS applications combined with the fact that computerized entities are embedded in the physical word. In this context, online debugging techniques are helpful because the debugger is connected to the device when an exception or crash occurs.

This paper reports on our experiences on applying two different families of remote debugging techniques for CPS systems: traditional remote debugging (cf. Section 3) and a novel debug technique we call out-of-place debugging (cf. Section 4). Remote debugging allows developers to debug from another machine than the one where the target application runs. In particular, we study online remote debugging which typically offers the same functionalities than online debuggers (*i.e.* breakpoints and step-by-step execution) while performing them from a remote machine by means of *e.g.* remote proxies. This experience report shows how we use these two remote debugging with a real CPS implemented in Pharo. Our evaluation (cf. Section 5) shows that out-of-place debugging is a promising technique to debug CPS systems as it improves the debugging experience by considerably minimising network exchanges.

2. Motivation

Debugging CPS systems is hard. Indeed the complexity of distributed applications together with the lack of control of the physical world makes bugs dependent on the context and hard to reproduce. To illustrate these problems, in this section we introduce a case study application in the domain of energy monitoring. This case study presents sporadic failures that seem at first sight hardware-related. We then present several debugging techniques and briefly discuss how they address this application scenario. This discussion motivates the choice of debugging techniques compared in this article.

2.1 Case Study: Sensor Monitoring App

Our case study is a CPS that monitors the temperature of a room. We called this case study *Sensor Monitoring App*. This monitoring system is made of a small computer (referred to as the device) connected to a temperature sensor and an LCD screen. We deploy the device in the room that we are interested to monitor. The sensor probes the room's temperature and displays the result on the LCD screen. This device is connected to the network via WiFi or ethernet and is configured to send alarms to the end-user if the temperature of the room exceeds a given level (*e.g* in a food storage room). The internet connection is bi-directional: the device can also receive updates such as user configuration and firmware updates.

Architecture. The application's architecture is illustrated in Figure 1. We built this monitoring system using a *Raspberry Pi* computer [1] together with a *GrovePi* board[2]. The Raspberry Pi is a cheap but powerful small computer that can run a Linux operating system and can be extended with standard sensors and actuators such as humidity sensors or screens. The GrovePi board has been designed to be put on top of the Raspberry Pi and provide easy access to a variety of sensors. GrovePi already ships with specific drivers written in different programming languages. Our monitoring system controls the hardware using a Pharo API to the Python driver of the board, to ask for sensor input and display data on the LCD screen. The application's architecture is illustrated in Figure 1.

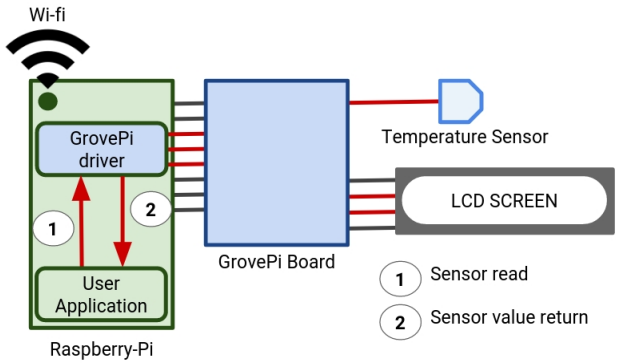

Figure 1. CPS scenario. A Raspberry Pi with a GrovePi board controls LEDs, a temperature sensor and an LCD screen. The developer remotely deploys, tests and interacts with the application from his development machine through WiFi.

Our monitoring software installed in the device is written as a *Pharo* [BDN+09] application. This application queries the GrovePi driver for the current temperature with a frequency of 2 Hertz. The driver performs a sensor read and returns a string value of it. Our application then converts the obtained string to a number and shows it in an LCD screen. Our application is also configured with a max temperature threshold. While the sensed temperature is lower than the configured threshold, a green LED is turned on. Otherwise, the application sends an alarm to the end-user. We implemented this alarm for our scenario as a red LED turning on.

It is worth to note that we take into account in Sensor Monitoring App that *i/o* errors can occur and that sometimes the sensor coming from the data could be erratic. As such, the application validates the value obtained by the sensor before treating it (*i.e.* checks the value is not a null string).

Development process. Several options exist to deploy, update and launch our application in the device. This can be

[1] https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[2] https://www.dexterindustries.com/site/?product=grovepi-starter-kit-raspberry-pi

done remotely from the developer's computer. The developer must have network access to the device to be able to do any maintenance task. As a result, the development and debugging cycles are much slower since deploying and testing a change in the application requires connecting to the device, transferring the program, installing and testing. Any new bug fix that needs to be tested on the real device requires restarting this cycle.

Alternatively, a developer may develop *in-place*, *i.e.* directly in the device. This speeds-up the development cycle by removing remote program transfers. However, this typically requires physical access to the device to plug a screen, mouse and keyboard which is not always possible in a CPS system.

Finally, the developer could opt for a hybrid approach that maximizes the work done on the local development machine. In this way, the developer will delay the deployment and testing in the real device until he/she believes that the code is correct. The main drawback of this approach is that at deploy time many unanticipated issues could appear due to the late integration and testing with the real hardware.

The bug. When testing our application, the device works fine the majority of the time. However, from time to time false alarms are sent to the user. Restarting the device solves temporarily the problem: after an undetermined period of time the bug reappears. Reproducing the bug is not easy because we cannot predict the timing of the bug. In addition, in production mode the temperature monitor works remotely, so when there is a problem we cannot know for sure what is happening. Reproducing the exact conditions under which the bug happens is complicated.

The Sensor Monitoring App illustrates the core problems when debugging CPS systems. The main problems steams from the fact that CPS systems combine the challenges of both debugging distributed and embedded systems: they depend on a sequence of events that may happen in unexpected timings and depend on the external physical environment. This makes bugs hard to reproduce, and thus difficult to diagnose and fix.

2.2 Debugging techniques overview

In this section we discuss debugging techniques that can be employed for CPS Systems. Since CPS systems are still in their infancy, we review general-purpose debugging techniques, and later discuss the closest related work in distributed debugging. We classify debugging support in (1) techniques that simulate or emulate hardware (called *pre-deployment* in [TS12]) and (2) online and offline debuggers (called *post-development* tools in [TS12]).

Simulators. Simulating the CPS allows the developer to experiment scenarios to study the CPS or to reproduce bugs [TS12]. While debugging, the developer can fully benefit from an offline environment and simulate input from the hardware, *e.g.,* sensor reads. It becomes possible to experiment with bug fixes until a solution is found. Then the corrected code must be put in place in the real CPS. Simulating a CPS requires to fully understand and model both the CPS and the cause of the bug [TS12]. Otherwise, it is complicated to reproduce the non-determinism causing the bug. The main drawback of simulators is that if they do not accurately capture the behaviour of the CPS to anticipate and reproduce the problem, unanticipated problems can happen in the productive environment after deployment.

Offline debuggers. When debugging non-deterministic systems like CPS, a complete view of the system is necessary to analyze the state that made the program crash. This could be achieved by extracting *core dumps*, *i.e.,* a snapshot of the whole state of the program when a crash happens, and providing interpreters for these dumps [MM80]. However, a core dump does not provide enough information to debug high-level programs as it only provides the state of the memory and a call-stack. These two elements are not sufficient in many occasions, since they miss other contextual information needed to totally understand the nature of a bug. For example, the value of the function arguments are not provided, nor the values referenced in the different levels of the stack.

Alternatively, one can turn to *replay* debuggers [MH89] in which the debugger records a trace of relevant events of a program and re-executes them in a debugging session. The debugger offers online debugging primitives such as breakpoint and stepping to examine the recorded state of the program without altering its behaviour. However, the overhead introduced to produce a trace of the execution is high [MH89]. The fault is debugged when the system finished running. This means that applications that need to analyse large quantities of data could produce hours of lost computations [GIY+16].

Online debuggers. Opposite to offline debuggers, *online debuggers*, often called *breakpoint-based debuggers*, control the execution of the program and interact with it through operations like pausing/resuming execution and step-by-step execution. Among online debuggers we can find *remote debuggers*, which particularly suit CPS systems. In fact remote debuggers allow developers to remotely connect to a running application and actively debug it.

When an exception is raised, a debug session is started on the CPS but the debugger window opens on the developer's computer. One can debug the program as if it was running on its own machine except every action is performed remotely on the CPS device. Remote debuggers are widespread in general-purpose mainstream languages. Examples of them are: JPDA [Ora17] provides remote debugging support for Java programs, the Visual Studio debugger [Mic17] for .NET applications, GDB [FSF17] for languages of the C family, and Mercury [PBF+15] for Pharo applications.

Remote debugging is a promising technique in the context of CPS since it is difficult to physically access a device and those devices may not have commodities for debugging like GUIs, mouse or keyboards. However, similarly to classic online debuggers, remote debuggers suspends the execution of an application when a breakpoint or a failure is encountered. This execution remains suspended until the execution is resumed manually by the developer. In a CPS system, it may not be possible to stop all the nodes participating in the system. In addition, the procedure can be error-prone as it is invasive, *i.e.,* modifications are directly applied in the system. Finally, remote debuggers require a constant connection with the remote device: if the connection is lost, unfinished modifications can lead to non-deterministic behaviour.

Alternatively, out-of-place debugging could be employed for CPS systems [Mar17]. This is a novel debugging technique explored by some of the authors in prior work which allows to do online debugging on a program while avoiding the suspension of the *overall* application's execution. An out-of-place debugger allows to remotely debug an exception on a machine external to the CPS by transferring the entire runtime information required to debug to the developer's machine. This allows developers to work on a complete debug session without affecting the debugged application. A developer can finally deploy all changes required to fix the bug as a single final commit step and resume the execution. These changes are propagated to all the debugged machines.

Distributed Debugging. A bulk of related work in distributed debugging has focused on Wireless Sensor Networks (WSN), the closest distributed systems to CPS [TS12]. Existing debugging for WSN are mainly designed to understand and reproduce the conditions of a bug [TS12]. However, they are typically limited to one aspect of debugging at a time; the addressed concerns are mainly monitoring, record and replay facilities and event analysis. Such systems would benefit from an online debugger to investigate and fix bugs at runtime.

2.3 Conclusion

From our literature review, we argue that online debugging techniques are better suited for CPS systems since they potentially could minimize the need for reproducing the bug: the debugger can be connected at the moment of the bug and then capture the runtime environment information of the bug as it manifests. The developer has access to the application's state at the exact moment the problem was perceived, significantly simplifying the debugging task. Since in CPS a bug can manifest in devices in which we cannot employ facilities to support debugging like a screen, keyboard, or mouse, we also argue that remote online debuggers are better suited than classical interactive online debuggers. Alternatively, out-of-space debugging could be also a good approach for CPS debugging since it allows to debug remote programs as if they are running in the local developer's machine, and then only a bugfix when it is considered properly tested.

In the remainder of this paper, we analyse both traditional remote and out-of-place debugging in the context of the Sensor Monitoring App. Before delving into the comparison of both approaches, we provide further details on PharmIDE and IDRA, the two concrete debugging tools used in our study which implement remote and out-of-place debugging for Pharo, respectively.

3. Remote Debugging in Pharo

In Pharo, developers can remotely debug a running application using the PharmIDE [Kud17], an implementation of the Mercury debugging model [PBF+15] which is now part of the Pharo distribution. PharmIDE offers online debugging with breakpoints and stepping commands and keeps a complete view of the stack, which is represented by an accessible object. It also supports restarting the program's execution from a particular context of the stack, hot-swapping the updated code when necessary. Mercury uses mirrors [BU04] to access objects in the debugged machine from the developer's machine. Applying a code change to a mirror immediately transfers such change to the debugged application and applies it in the debugged exception.

The general architecture of PharmIDE consists of two parts:

- A server running on the debugged Pharo application
- A client running on the developer's computer

A client can connect at any time to the remote running program and start a debugging session. When an uncaught exception happens, a debugger GUI spawns in the client side. This debugger GUI is the same one used to debug local processes. The difference between a local and a remote debugging session is that in a remote session every action happens on the remote side. The debugger allows then to inspect remote objects, to change remote code and perform all debugging operations (restart, step-over, step-through) on the debugged program. As in a classic debugger, it is possible to manually skip the faulting code to allow the execution to resume, but it is not possible to use it as a permanent workaround.

Note that every action performed through the debugger requires a network transfer. Changing a value, committing code modifications or stepping through an instruction sends a request to the debugged application and triggers data exchanges. Practical use of PharmIDE directly depends on the performance of the remote system, on which every debug operation is executed, and on the available network bandwidth.

4. Out-of-place debugging in Pharo

This section presents an overview of IDRA, an out-of-place debugger that the authors implemented in Pharo [Mar17]. In

a nutshell, IDRA supports online debugging by transferring the execution state of the debugged application to the local developer's machine. The developer proceeds then to debug as if the application was originally a local application. The remote application can then continue executing the next task that should process. Note that in the Sensor Monitoring App, however, there is only one task being executed, as such, the remote application is suspended after transferring the execution state to the remote machine.

IDRA debugging cycle. When a program running under IDRA throws an exception or stops in a breakpoint, IDRA serializes the program execution state and transfers it to the developer's machine. The developer can then proceed to debug locally an exact copy of the original program at the moment of the exception. If the developer discovers the cause, he can modify the application code locally to create a bugfix. At any time, the developer can decide to send all the changes of a bugfix in a single *commit* step to the debugged application. Finally, whether the developer submitted a fix or not, it is possible to resume the execution of the Sensor Monitoring App running on the device from the point it failed.

IDRA aims to provide a faster debugging cycle than PharmIDE since every debugging operation runs locally except for:

1. The initial start of the session that requires the transfer of the program state.
2. The final commit and proceed operations that require the transfer of the changes made locally.

IDRA allows the developer to modify and explore possible solutions locally without affecting in the deployed application that keeps running. IDRA is thus designed to require less network roundtrips during debugging exposing the same performance as debugging a local process with the single overcome of a slower startup and commit phases.

General Architecture. Figure 2 shows the overall architecture of IDRA consisting of two components: the IDRA Monitor (running as part of the debugged process) and the IDRA Manager (running in the developer's machine). The IDRA Monitor controls the target application and suspends its execution when it finds a breakpoint. Every time a breakpoint is hit, the IDRA Monitor serializes the stack trace and all objects reachable from it using the Fuel serializer [DPDA11] and sends it to the IDRA Manager. Fuel will serialize the contexts of the call stack and all the objects referenced from each context. The full state of these objects will be serialized in order to properly reconstruct the state. However, classes, global variables and the instances of IDRA are not included in the serialization.

On the developer's machine, the IDRA Manager deserializes the stack traces and opens a local Pharo debugger GUI on it. The developer's machine has a local copy of the stack at the moment the bug was produced, *i.e.,* no proxies to the remote machine are used at all. While the developer interacts with the application to produce a bug fix, all interactions are recorded by the Epicea changes logger [DCD13]. Finally, when a bug-fix is ready, all changes logged are serialized using Fuel and sent to the IDRA Monitor to be applied.

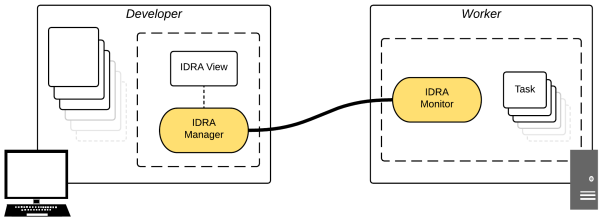

Figure 2. Representation of IDRA instances, manager and monitor, in a distributed system of two machines.

5. Comparing Remote and Out-of-place Debugging in the Sensor Monitoring App

In order to compare PharmIDE and IDRA, we benchmarked several isolated scenarios. The goal of the comparison is to determine the performance differences between both solutions.While remote debugging incurs on network overhead through the whole debugging session, out-of-place requires capturing once the whole application's state and transmitting it over the network. In the remainder of this section, we explain the benchmark setup we did on the Sensor Monitoring App and the results we obtained.

5.1 Debugging the sensor Monitoring App

We tried to solve the bug using PharmIDE and IDRA as debuggers. With both debuggers we have been able to spot the problem: sometimes the temperature sensor fails to read a correct value and cannot provide input to the querying software (*i.e.* our application). This bug seems random and is difficult to reproduce. However when the application runs for a significant amount of time, the bug appears from time to time and randomly produces exceptions. These exceptions are non-blocking but until they are solved, the application cannot be trusted by the user as it sends false alarms. In that case the inspected values in the debugger showed that the sensor returned a string with a specific value *"nan"* which meant that the returned sensor input was *"not a number"*. Our program was not entirely following the hardware's specifications, and we were looking for *null* values to handle invalid sensor inputs instead of this *"nan"* string.

Since our model was wrong, this particular problem may never have been reproduced in a simulation if we assumed that invalid inputs were *null* values. As such invalid inputs are rare, the effort to reproduce the bug can be huge as we did not know what we were looking for. In addition, on a so small and simple CPS device like the sensor applica-

tion, having runtime logging capabilities to trace the problem is not granted. Remotely debugging the application was of great help to catch the problem at the time it did appear.

We have been able to fix the bug with both debuggers, although the methodologies did differ. For example, in PharmIDE we remotely changed the code which was running on the Raspberry Pi. In this case we risk introducing new bugs that could make the device crash. However, we were able to query sensor inputs to test our modifications during the debugging process. Using PharmIDE we also experienced different performance problems, that were leading to the debugger hanging making it impossible to continue debugging.

With IDRA, we could experiment and validate changes before sending them to the system, but we were unable to request new sensor input as everything was done locally on the developer's machine.

5.2 Benchmarks Setup

To benchmark both scenarios we use the *Sensor Monitoring App*.

To run our benchmarks we used two different machines:

Developer Machine. Intel Core i7 6700HQ @2.60GHz x 8 with Intel Turbo Boost, 16 GB DDR4 RAM, Linux Mint 18.1 Serena - 64 bit, Pharo 6.0 #60499.

Raspberry Pi. ARMv8 quad-core @ 1.2GHz, 1 GB DDR3 RAM, Raspbian GNU/Linux 8 (jessie), Pharo 6.0 #60499.

We deployed the *Sensor Monitoring App* on a Raspberry Pi, and we let it run long enough to generate the required number of exceptions for each benchmark. The Raspberry was connected to a local network through a 100mbps ethernet connection. For each benchmark, we activated the Pharo Remote Debugger and the IDRA out-of-place debugger on a remote computer connected to the same network through a 100mbps ethernet connection.

Whenever an exception was raised, debuggers opened on the remote computer for the debugging activity. This setup was used to perform the benchmarks and to compare both debuggers.

5.2.1 Benchmark 1 - Session initialization

This benchmark measures how much time it takes for each debugger to open a debugging session on a given exception.

Benchmarking Methodology. This benchmark measures how much time passes between the developer's machine receives an exception and a debugger is opened for it. The Pharo remote debugger opens immediately a debugger for each exception it receives. On the other hand, IDRA opens only one debugging session at a time. Thus, to have an equivalent evaluation, we made sure to close each debugging session opened by IDRA before opening another one. We do not consider the time needed by the user interface to open a debugger GUI as both debuggers use a classic Pharo debugger session.

Results. We can observe that, on average, the Pharo Remote debugger is between a thousand and ten thousand times faster than IDRA. Figure 3 shows a boxplot of the results. The time is calculated in milliseconds. In fact the Pharo remote debugger takes approximately 15 μs on average, while IDRA takes around 60 ms. A 60 ms delay is however hardly noticeable to a developer's eye, and moreover it happens only once during the debugging life-cycle, when the session is opened.

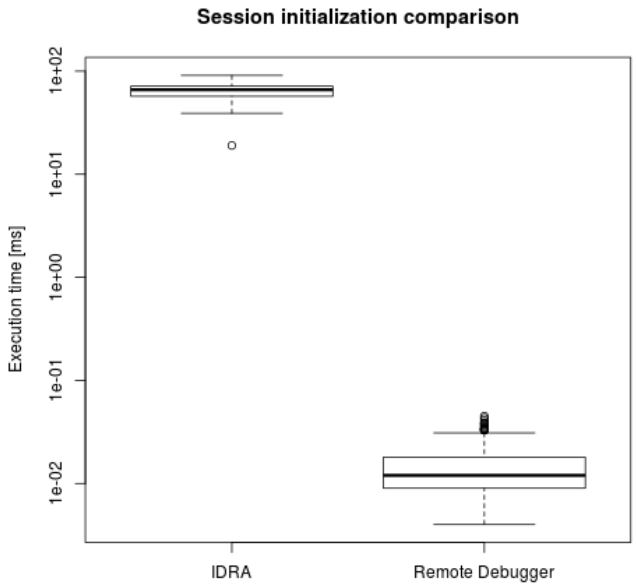

Figure 3. Boxplot of the session initialization time for an exception.

This result is expected because of the way IDRA handles arriving exceptions: while the Pharo Remote Debugger immediately calls the user interface to generate a debugger, IDRA puts the received exception in a queue, then another thread reads from that queue, and asks the user interface to open a debugger. This thread reads on the queue every 60 ms, approximately the delay measured in this benchmark.

5.2.2 Benchmark 2 - Stepping operations

We performed a series of benchmarks to measure the time spent to execute the different stepping operations using each debugger. The operations we are interested in in this benchmark are:

Restart the execution from a selected point in the stack.

Step Into the next expression, shows the code corresponding to the method invoked in that line.

Step Over the next expression, executes the next line and goes to the following.

Step Through the next expression, executes parameters evaluated and steps into the proper code execution.

Proceed simply continues the execution not debugging.

Benchmarking Methodology. We wrote our benchmarks as follows:

1. We first **Restart** the execution from a point in the stack.
2. We then execute the operation that insterests us (**Step into/over/through**). When we measure **Restart**, we skip this step **Restart operation**.
3. Finally, **Proceed** the computation

This actions represent a typical debugging session, except by the fact that no code is changed. It is however consistent to evaluate the execution time of the operations on both debuggers.

Results. Our results show that IDRA is constantly faster than the Pharo Remote Debugger by more than one thousand times. Table 1 shows the result of this benchmark on the Sensor Monitoring App. The time was calculated in microseconds.

Operation	IDRA [μs]	PharmIDE [μs]	Speedup
Step Into	372.1	1353378.4	3600x
Step Over	345.1	1571287.6	4500x
Step Through	353.6	1378951.5	3900x
Restart	362.5	1044374.9	2800x

Table 1. Execution time of single debugging operations (on average) on the Sensor Monitoring App.

When IDRA handles a remote exception, the exception and all the stack information is copied and sent to the developer's machine. A debugging session is always opened on a local copy of the exception (and its stack), which makes the debugging session a normal *local* Pharo debugging session.

On the other hand, the Pharo Remote Debugger reconstructs a remote exception by means of proxy objects of the exception itself and of the related stack. The debugging operations will be executed on the remote machine, introducing communication and network overhead for each of the executed operations.

5.3 Benchmark 3: Network usage per exception

This benchmark measures how much data is sent (in bytes) between the two components of the debuggers. This means exchanging data between *monitor* and *manager* in the case of IDRA and *server* and *client* in the case of the Pharo remote debugger.

Benchmarking Methodology. We measure this overhead in different ways, depending on the debugger. In IDRA we measure the size of the data received on the IDRA Manager, since we have control over the TCP connection. On the other hand, the Pharo Remote Debugger uses the library Seamless [PBF+15] to handle the TCP communication. To assess how much data is exchanged through Seamless, we use a logger provided by the framework, which returns detailed statistics over all the communication that happened since it was started.

This benchmark measures this transfer size of each debugger when transferring 0, 10, 20, 30, 40 and 50 exceptions. It gives an idea on how much communication time is needed to transfer an exception using both debuggers. The communication time is not evaluated since it depends on the network and it would require a notion of distributed clocks to be correctly evaluated. This communication time can be inferred knowing the amount of data transferred and the communication speed of the network.

Results. Our results show that the data exchanged per exception is significantly higher in the case of IDRA. Figure 4 shows the number of exchanged bytes with different number of exceptions. The x-axis shows the number of exceptions and the y-axis the number of bytes exchanged. The y-axis is displayed with a logarithmic scale.

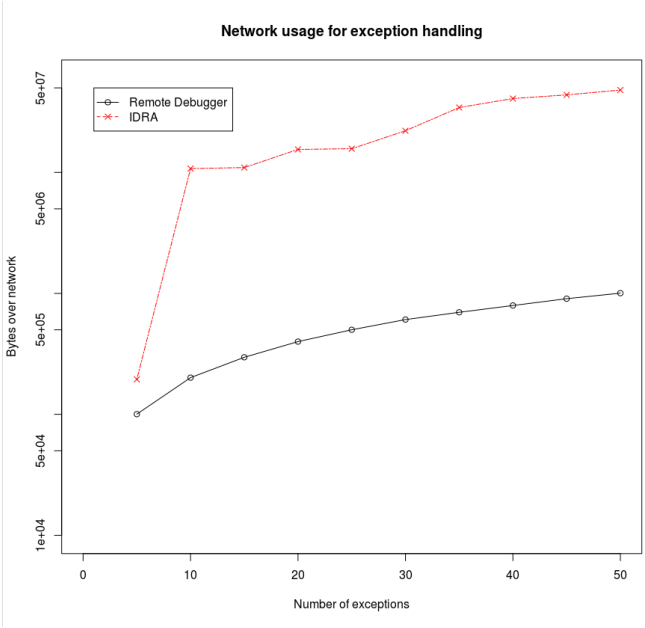

Figure 4. Plot of the number of bytes exchanged for an increasing number of exceptions.

For each exception, in a constant way when increasing the number of exceptions, IDRA exchanges over network ten to one hundred times more bytes than the Pharo Remote Debugger. The two curves mostly have the same increasing, similar to logarithmic. However, as it is clearly visible, between five and ten exceptions IDRA consumes a considerably higher number of bytes. At the moment of writing, this entropy is believed to be due to a bug of the serialization graph produced by Fuel. Moreover, we obtained these results

in a IDRA prototype that has no optimizations. We expect to improve these results in further work.

5.4 Benchmark 4: Network usage per code change

This benchmark evaluates the size of code changes transferred to the debugged machine.

Benchmarking methodology. We analyzed the size in bytes, in the means of network communication, of transferring a change from the developer's machine to the debugged machine. The changes that interest us are the following:

- **No operation**: no changes are made. A browser is opened and changes are sent.
- **A class addition**: a class named `Test01` is added to the default package.
- **An instance variable addition**: an instance variable named `instanceVariable` is added to `Test01`.
- **A class variable addition**: a class variable named `classVariable` is added to `Test01`
- **A method code change**: a method of the class `Test01` is changed adding a line of code.

The way we measured this data for each debugger differs because both debuggers do not behave similarly. In IDRA, changes happen locally and are then sent to the remote machine through the IDRA Changes Handler. The changes are applied in the remote machine only when the user explicitly calls this functionality. The Pharo Remote Debugger, on the other hand, applies directly the changes on the debugged machine. This is why all the operations are evaluated only after opening the browser and browsing to the right class.

We consider and measure also the transfer of a *no operation*. In IDRA, a *no operation* means to commit an empty list of changes. In the Pharo Remote Debugger it means the operation of opening opening a browser without doing any modification.

Results Figure 5 shows the network usage in bytes for each operation, the y-axis uses a logarithmic scale. Our results show that IDRA uses eight to ten times less network when compared to the Pharo Remote Debugger for simple committing operations. The only exception to this is the *no operation* case where there is no evident difference.

We believe that these results are due to the fact that the Pharo Remote Debugger uses a remote browser, which contains proxies to many entities of the remote image. Every modification constantly generates a request to the remote image to update, which does not happen in IDRA because the changes are applied to the local code base.

5.5 Discussion

Our IDRA prototype shows promising results for CPS debugging. Indeed, it has all the features of an online debugger: breakpoints, stepping operations, the ability to inspect and interact with the executing program. Moreover, it gives developers the illusion of working as a regular local debugger while indeed it debugs a remote process. This feature also isolates the developer's environment from the debugged environment, giving live programming developers the freedom to explore several solutions to the bug before committing one. Generally speaking, IDRA is a complete debugger achieving good performance.

We discuss in the following points the applicability, limitations and point of improvement of IDRA.

Applicability. The benchmarks show that IDRA is a promising alternative to PharmIDE, especially when comparing the execution time of single debugging operations. Since we are talking about remote debuggers which communicate over network, the speed of the network can play a decisive role in choosing one of the two debuggers. In fact, if the network imposes less overhead, it might be convenient to use one or the other depending, for example, on the stack size. In this respect one needs to consider that IDRA is only a research prototype, and does not present any optimization. As part of the future work we plan on creating concrete guidelines to help developers choosing between classic remote debugging and out-of-place debugging.

External Resources. External resources such as local files or sensors at the local machine's hardware are not shared between the IDRA Manager and the IDRA Monitor. This means that a developer using IDRA has to avoid the access to sensors from her machine because that resource is not locally available. This problem does not appear in the remote debugger because the remote resources can usually be accessed through proxies. However, in our solution we reconstruct an environment on a separate machine. This problem is akin to code mobility, and many possible solutions can be found in the literature [FPV98].

Control Serialized Objects. Fuel [DPDA11] allows the serialization of all objects reachable from a certain starting object, exceptions and stack traces comprised. While this is a desirable property for a serializer, this may become problematic when employing it for an out-of-place debugger. Indeed, sometimes the reachable object graph may have references to global objects and/or include objects that the developer did not expect. This has a direct impact on the size of the exchanged stack traces.

A possible solution for this issue would be to analyze and optimize the serialization process for our case. For example in the current implementation of the Pharo Remote Debugger [Kud17], the serialization process is optimized to serialize proxies over the TCP network.

Network stability. Communication in IDRA happens over TCP/IP and we rely on its mechanisms for network failure handling. This means a slow network or machine can lead to errors because of too short timeouts. IDRA does not include so far any high level failure handling, nor a robust re-connection mechanism either.

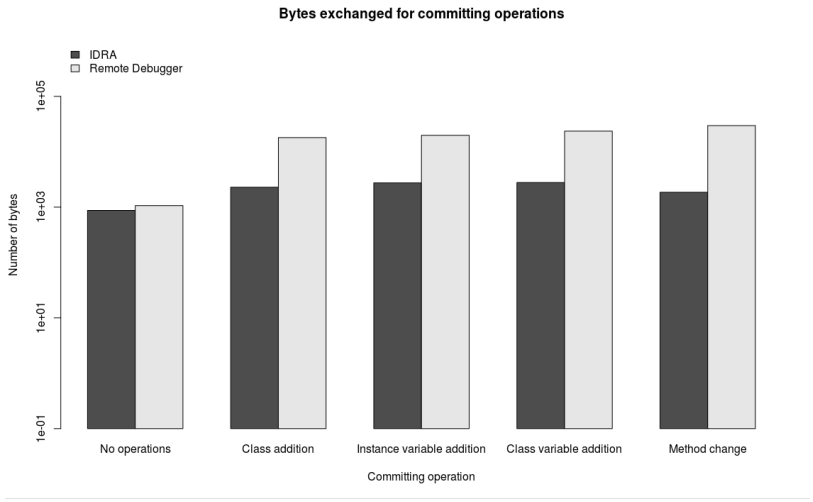

Figure 5. Bar plot of the bytes exchanged to commit one change.

Concurrency. Out-of-place debugging does not explicitly support multi-threading. Concurrent processes will not be handled in a particular way, and eventual state changes are not considered. However, this limitation is also present in classic debuggers, including the compared PharmIDE.

6. Conclusion

Cyber-Physical Systems (CPS) are an emerging kind of distributed systems that introduce hardware components to sense and interact their surrounding environment. In this paper, we studied debugging support for CPS written in Pharo. Reproducing errors due to unpredictable and live environments is particularly hard. All the runtime information from the moment when the bug shows itself is lost if the debugging activity does not happen at that exact moment.

This paper reported our experiences with using two remote online debuggers available in Pharo for debugging CPS: the Remote Pharo debugger named PharmIDE and a novel *out-of-place* debugger IDRA. PharmIDE allows to debug the running application directly on the device. All debug operations are performed on the remote device but from the developer's computer. IDRA, on the other hand, is an *out-of-place* debugger which allows the developer to debug a remote program on his own computer instead of directly on the device. Debug actions are performed locally and all code changes are later committed to the remote device.

We studied both Pharo remote and out-of-place debuggers in the context of the Sensor Monitoring App, a temperature monitoring application written in Pharo. The application uses a driver to access the temperature sensor and from time to time the recovered value provokes an exception. This is due to a sensor error, which conditions are very hard to reproduce. Our evaluation of both approaches showed that IDRA provides faster performances in terms of debugging operations speed and consumes less network data when committing code changes. On the other hand, the Pharo Remote Debugger is faster to initialize debugging sessions, mainly because Idra uses an exception queue-handling mechanism. The Pharo Remote Debugger also consumes significantly less data when exchanging exceptions on the network, which is probably due to the non-optimized underlying serialization mechanism of IDRA.

Having faster debugging operation improves the user experience, since the user would not need to wait several seconds for an operation to be executed (i.e. it does not affect the remote execution context). Considering a live programming environment, IDRA also allows to locally test a solution before deploying the code. This is not possible using classic remote debuggers like the Pharo Remote Debugger.

Acknowledgments

We thank the anonymous reviewers for their comments.

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 331–344, New York, NY, USA, 2004. ACM.

[DCD13] Martin Dias, Damien Cassou, and Stéphane Ducasse. Representing code history with development environment events. *CoRR*, abs/1309.4334, 2013.

[DPDA11] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Clustered serialization

with fuel. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 1:1–1:13, New York, NY, USA, 2011. ACM.

[FPV98] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[FSF17] Inc. Free Software Foundation. The gnu project debugger. `https://www.gnu.org/software/gdb/`, 2017. Accessed: 2017-04-14.

[GIY+16] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. Bigdebug: Debugging primitives for interactive big data processing in spark. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 784–795, New York, NY, USA, 2016. ACM.

[Kud17] Denis Kudriashov. Pharmide: Pharo remote ide to develop farm of pharo images remotely. http://dionisiydk.blogspot.be/2017/01/pharmide-pharo-remote-ide-to-develop.html, 2017. Accessed: 2017-05-10.

[Mar17] Matteo Marra. IDRA: An out-of-place debugger for non-stoppable applications. 2017.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, December 1989.

[Mic17] Microsoft. Remote debugging. `https://msdn.microsoft.com/en-us/library/y7f5zaaa.aspx`, 2017. Accessed: 2017-05-12.

[MM80] D. R. McGregor and J. R. Malone. Stabdumpa dump interpreter program to assist debugging. *Software: Practice and Experience*, 10(4):329–332, 1980.

[Ora17] Oracle. Jpda - java platform debugger architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/, 2017. Accessed: 2017-05-10.

[PBF+15] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, and Marcus Denker. Mercury: Properties and design of a remote debugging solution using reflection. *The Journal of Object Technology*, 14(2):1:1, 2015.

[TS12] Sreedevi T.R., , and Mary Priya Sebastian. A classification of the debugging techniques of wireless sensor networks. *2012 International Conference on Advances in Computing and Communications*, 00:51–57, 2012.