Crowdsourcing GUI Tests

Eelco Dolstra*, Raynor Vliegendhart† and Johan Pouwelse‡
* *LogicBlox, Inc., Atlanta, GA, USA,* eelco.dolstra@logicblox.com
† *Department of Intelligent Systems, Delft University of Technology, Netherlands,* r.vliegendhart@tudelft.nl
‡ *Department of Software and Computer Technology, Delft University of Technology, Netherlands,* j.a.pouwelse@tudelft.nl

***Abstract*—Graphical user interfaces are difficult to test: automated tests are hard to create and maintain, while manual tests are time-consuming, expensive and hard to integrate in a continuous testing process. In this paper, we show that it is possible to *crowdsource* GUI tests, that is, to outsource them to individuals drawn from a large pool of workers on the Internet, by instantiating virtual machines (VMs) running the system under test and letting testers access the VMs through their web browsers. This enables semi-automated continuous testing of GUIs and usability experiments with large numbers of participants at low cost. Several large experiments on the Amazon Mechanical Turk demonstrate that our approach is technically feasible and sufficiently reliable.**

I. Introduction

Testing of graphical user interfaces (GUIs) is a perennially difficult problem. Ideally, developers test a GUI automatically, just as any other part of a program; this allows a GUI to be tested from a continuous build system, e.g., on every commit. However, automated GUI testing approaches tend to be brittle: test cases can easily break due to minor changes in the GUI, leading to high test maintenance effort or bit rot in the test suite. It is also difficult for a computer to determine if the visual appearance of a program is correct.

Thus, GUI testing remains primarily a human task: flesh-and-blood testers are required to execute test actions and check the results. This is labour-intensive and expensive. For instance, it is hard to expect developers to perform an in-depth GUI test on every commit. And employing dedicated testers is *inelastic*: it is hard to quickly scale the number of testers up or down in response to changes in demand (e.g. to continuously test a new experimental branch of the product).

Because of this, companies have applied *crowdsourcing* to GUI testing: i.e., they outsource GUI testing to a large pool of testers around the world. However, these approaches require testers to install and maintain the application under test on their local machines. This imposes a high barrier to entry to testers and increases the cost of testing.

In this paper, we show that it is possible to crowdsource GUI tests using automatically instantiated *virtual machines* (VMs) that testers access through their web browsers. This moves the burden of maintaining the test environment from the tester to an automated process on a server. We have implemented a prototype implementation of this idea on Amazon's *Mechanical Turk* (MTurk), a crowdsourcing marketplace that allows *requesters* to submit *Human Intelligence Tasks* (HITs) to be performed by *workers* against a fee. When workers accept a GUI testing task through the MTurk web site, they are presented with a web page that shows the display of a remote VM running the GUI under test and allows mouse and keyboard interaction with the VM. The workers are asked to execute a sequence of steps described in the task and report the results. The interaction of the testers with the virtual machines is captured by recording the displays of the VMs, allowing developers to analyse and reproduce reported problems in a much more straightforward manner than, say, from a problem description in a bug report.

This approach has two primary applications explored in this paper. The first is semi-automated *continuous testing*: periodically, or every time a developer commits a change to the source of a project, a continuous build system compiles the latest revision of the project, then creates HITs in Mechanical Turk to test the project. The second is *usability studies*: a HIT can ask workers to accomplish a goal, rather than perform a precisely described sequence of steps. Developers can then draw qualitative and quantitative conclusions about the usability of their program by observing success rates and completion times, and analysing recordings to discover interesting interaction patterns. Conventionally, such experiments are difficult because it is hard to find a sufficiently large number of participants to allow statistically significant conclusions to be drawn; crowdsourcing makes it much easier to mobilise a large group of participants quickly.

There are two economic arguments to crowdsource GUI tests. First, as with conventional outsourcing, testers often come from lower-income regions and are thus likely to be cheaper than local hires. Second, the labour pool is much more flexible: it is easy to scale up or down the number of testers as conditions require.

To evaluate our approach, we have performed several experiments involving 398 workers. To determine feasibility of crowdsourcing for continuous testing, we used a number of test subjects: KDE and Xfce, two desktop environments for Unix, and Tribler, a peer-to-peer file-sharing program. This evaluation sought to answer two principal research questions: 1) *Is crowdsourcing of GUI tests technically feasible?* (For instance, bandwidth or latency limitations of workers might make it too hard to complete tasks in a reasonable amount of time.) 2) *Is the method sufficiently reliable?* (E.g., if the false negative rate is too high, bugs go undetected; if the false positive rate is too high, developers will waste time investigating non-existent problems.) Our experiments suggest that both are the case, though further work on HIT design is desirable to improve reliability.

For crowdsourcing of usability experiments, the main research question is whether it is possible to get statistically significant measurements of task completion times, despite the huge variance in worker connection speeds. Therefore, we ran *A/B tests* of variants of Tribler to determine whether an experimental user interface feature had a measurable effect on user efficiency. This experiment suggests that crowdsourcing is a cheap and effective method to run usability experiments involving hundreds of participants.

The structure of this paper is as follows. We first describe the background of our work (Section II). We then give a high-level overview of the method (Section III) and discuss technical aspects of our prototype system (Section IV). Finally, we present and discuss the results of our experimental evaluation (Section V).

II. Background and Related Work

A. GUI Testing

Graphical user interfaces are an important part of many software systems, such as desktop or mobile applications. The GUI is typically the most visible part of an application to end users. Thus, developing and testing the GUI takes up a significant part of the development effort — as much as 50–60% [1]. However, testing a GUI is a difficult problem because it is hard to automate. As with most other forms of testing, automating GUI tests is desirable because it gives developers confidence that the changes they make do not break functionality (e.g., by having a continuous build system run a test suite on every code change). Unfortunately, it is difficult to create a test case for some GUI functionality because it is hard to conveniently specify the required *input* (e.g., mouse clicks) and the expected *output* (e.g., the desired appearance of the application window).

Automated GUI testing approaches range from low-level, *capture/replay* methods to high-level, model-driven methods. In the former, a testing tool records the keyboard and mouse events from a sample session performed by a developer. These events can then be replayed to test the GUI. The downside of this approach is that it is very sensitive to changes to the GUI: in the most simplistic case, where one records (say) mouse clicks using their absolute screen coordinates, a minor rearrangement of GUI elements can cause the test to fail. Using the identity of the logical GUI element that received the event makes the test less brittle, but does not work if the absolute position *is* significant. Model-driven approaches (see e.g. [2], [3]) use white-box knowledge of the internal structure of the GUI to generate test cases. Model-driven tools tend to be specific to a programming language and GUI framework. An important issue in automated testing is the test oracle problem: how does the test decide that the visual appearance of the GUI is correct? Doing a pixel-precise comparison of the screen or window against a reference output is clearly fragile. (The "pass" criterion for GUI tests is therefore often simply that the application under test did not crash.) Various approaches to automate test oracles are described in [4], [5], [6], [7].

There are also semi-automated methods; e.g., White and Almezen [8] propose a method that uses a finite-state model of the GUI to generate test cases to be performed by a human tester. In their evaluation, they distinguish between *defects* (deviations from the specification) and *surprises* (a departure from "expected" behaviour, as seen by the user). Clearly, the latter are hard to identify using an automated process; as Meyers noted, "Automated testing tools are rarely useful for [current GUIs], since they have difficulty pretending to be users" [9].

Thus, GUI testing remains for a large part a human activity: human testers use the GUI and check the results, on an *ad hoc* basis or by following a test plan. But as we noted above, this is expensive and inflexible in responding to changes in demand.

B. Crowdsourcing and the Amazon Mechanical Turk

Because of this, we explore a different approach: namely, to *crowdsource* the problem, which is the act of outsourcing a task to individuals recruited from a large pool of available workers on the Internet with whom one has no direct relationship [10]. As an early example, the *Distributed Proofreaders* website[1] of Project Gutenberg asks volunteers to proofread and correct automated scans of books for mistakes in the OCR process. Crowdsourcing activities vary wildly, ranging from large tasks such as website design (e.g., crowdSPRING[2]) to microtasks such as classifying images. Tasks are often performed in exchange for a fee, such as a "micropayment" of a few cents for small task.

The main attraction of crowdsourcing for task creators is the low cost and flexibility in recruiting participants. Crucially, crowdsourcing is *elastic*: increasing or decreasing the number of tasks in response to changing demand is much simpler than if one had to hire or fire workers.

The best-known example of a crowdsourcing system is Amazon Mechanical Turk[3] (MTurk), created by Amazon.com in 2005. The name originates from an 18th-century chess-playing "machine" that in reality contained a hidden person to decide chess moves. Analogously, MTurk allows one to submit tasks and get results back as if they were performed automatically. These tasks are called *Human Intelligence Tasks* (HITs); the submitters of HITs are called *requesters*, while individuals who perform tasks are *workers*. Typical tasks include classifying web pages or images (e.g.

[1] http://www.pgdp.net/
[2] http://www.crowdspring.com/
[3] http://www.mturk.com/

to screen for offensive content), transcribing audio fragments, translating text fragments, searching for company websites on Google to determine search engine presence, and, unfortunately, engaging in various forms of spam [11]. The reward is typically a few US dollar cents.

The MTurk workflow is as follows. Requesters create a HIT by writing a task description and a summary, and add keywords to allow workers to find it easily. The requester also sets the desired number of *assignments*, which is the number of workers that should perform the HIT, and the reward per assignment. The requester submits the HIT to MTurk, receiving a *HIT ID*. The HIT then becomes visible on the MTurk website to workers who view or search the list of available HITs. Workers can preview and then *accept* the HIT, at which time an assignment is instantiated with a unique *assignment ID*. The task description in a HIT can be an arbitrary HTML page; apart from a description of the steps to be performed by the worker, it typically contains some form fields to allow the worker to enter results. After completing the task and clicking on the submit button, the worker's answers are stored by Amazon and can be viewed or downloaded by the requester. The requester can then *approve* or *reject* an assignment; in the former case, the worker is paid the HIT's reward.

There has been a large number of publications on using MTurk in research. For instance, it has been used to create training data for machine translation systems [12], [13], the results similar to those obtained from professional translators but more than an order of magnitude cheaper; to train automatic speech recognition systems [14], finding that the ability to easily gather more data compensates for decreased data quality; and in social science and psychological experiments [15], [16], where the use of MTurk helps to recruit more participants than would otherwise be feasible, and the wider demographic range of workers reduces the typical selection bias caused by using only university students in such experiments. On the other hand, Adar [17] criticises crowdsourcing research, pointing out that "showing that humans can do human work is not a contribution."

More directly relevant to the present work is that MTurk is used for UI testing. Testing web applications is technically straightforward, since workers already have a web browser; thus a HIT for testing the usability or functionality of a web site simply provides the worker with a hyperlink to the website to be tested. Going further, TryMyUI[4] provides usability testing to its customers by recording both the screen of workers, showing them interacting with the website, and the worker's microphone, allowing them to comment on the website. TryMyUI recruits testers on MTurk; however, the screen and audio recording requires workers to run a special Java applet. uTest[5] (not MTurk-based) allows testing of desktop and mobile applications using crowdsourced testers. However, it requires them to install the application under test as well as tools for recording sessions on their own devices. By contrast, our approach uses remote virtual machines accessed through a standard web browser. Thus it has a much lower barrier to entry for testers, enabling cheaper crowdsourcing.

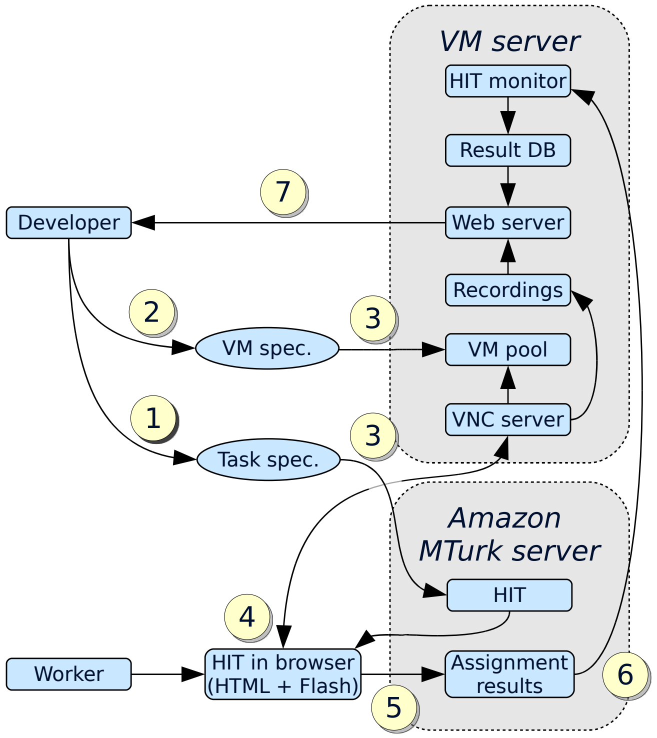


Figure 1. Overview of GUI testing on MTurk

III. Overview

This section gives a high-level overview of our MTurk-based GUI testing approach. The main technical goal is to allow MTurk workers to start testing GUIs with as little overhead as possible. Thus, the installation of specialised software on the worker's machine (such as the system under test) should not be necessary: everything should run in the worker's browser. We accomplish this by automatically instantiating *virtual machines* running the system under test on a server. The workers then connect to the display, keyboard and mouse of the virtual machine through a Flash application included in the HIT.

A schematic depiction of the process of creating a GUI testing HIT, performing the HIT assignments and processing the results, is shown in Figure 1. The steps in this process referenced in the figure are as follows:

① — The developer writes a *task description* that lists the steps to be performed by workers. This description is an XML file listing *steps* that consist of an *action* to be performed and a *question* to be answered by the worker.

[4]http://www.trymyui.com/
[5]http://www.utest.com/

```
<task description="Tribler download test">
 <step onFailGoTo="end">
  <question>Do you see a window named
   "Tribler"?</question>
 </step>
 <step onFailGoTo="channels">
  <action>In the search box, type "Ubuntu" and
   press enter. Wait a few seconds.</action>
  <question>Do results appear?</question>
 </step>
 <step onFailGoTo="channels">
  <action>Click on the Download button next
   to the top result. This should start
   the download.</action>
 </step>
 <step onFailGoTo="channels">
  <action>Click on the Library button at the
   top.</action> <question>Is the download in
   progress?</question>
 </step>
 <step id="channels" onFailGoTo="offensive">
  <action>Click on the Channels button at the
   top.</action> <question>Does a list of
   "Popular Channels" appear?</question>
 </step> …
 <step id="offensive">
  <question type="text">Click on the Home button
   at the top. If you see offensive words appear
   at the bottom of the window, list them here. …
 </step>
</task>
```

Figure 2. Task description of a Tribler GUI test

Questions can have several types, such as Boolean (yes/no) or textual (e.g. to ask an open question). It also specifies metadata such as the reward for the task.

Figure 2 shows a somewhat abbreviated task description for testing *Tribler*, a completely decentralised peer-to-peer file-sharing application [18]. The task tests Tribler's most significant functionality: the ability to search for and download files and to find and browse "channels" (a mechanism for users to publish collections of files). Because Tribler has an optional "family filter" to remove potentially offensive results from search results, the task also asks workers to look at the "network buzz" search terms and report any offensive words they see. This is a highly subjective task that can only be performed by a human.

If a step does not specify a question, the default Boolean question "Did you succeed?" is used. Actions may be omitted if the question just requires the worker to visually inspect the display. The onFailGoTo attribute is used to cause a failing step (i.e. one answered with "no") to skip to the indicated step. For instance, if searching for "Ubuntu" fails to return results, then the steps asking the worker to download Ubuntu will be skipped.

② — The developer also writes a *virtual machine specification* that describes how to build a virtual machine containing the necessary software automatically and reproducibly. (This is discussed in more detail in Section IV.) For instance, for the Tribler task, the VM specification states that the virtual machine should run the Xfce desktop environment (to provide a window manager, necessary in a Unix-based GUI environment), contain the Tribler client and start Tribler automatically when the VM boots.

The VM specification can provide an *acceptance check*, a script run after the worker finishes the task. Its purpose is to do a basic sanity check to verify that the worker completed the task correctly. This allows some suspicious assignments to be flagged automatically; e.g., if the acceptance check fails but the worker says that all steps succeeded, this may indicate cheating. For Tribler, the acceptance check verifies that a file has appeared in Tribler's download directory, indicating that the worker completed the download step correctly.

③ — From the task description and the VM specification, virtual machines are instantiated on the VM server, and a HIT is created in MTurk (using the MTurk API). The HIT is shown on the MTurk worker website.

④ — When a worker accepts a GUI testing hit in his web browser, a web page appears showing a virtual machine and the first step to be performed (along with some general instructions). Mouse movements, clicks and key presses are sent to the virtual machine. Answering a question causes the next step to appear. Figure 3 shows an example of a HIT containing a running VM. The virtual machine instance is unique to this particular worker; each worker gets his own VM instance, and thus cannot interfere with other workers. The VM server starts recording the screen of the VM as soon as the worker connects, and stops recording when the worker disconnects. The server also logs keyboard and mouse events.

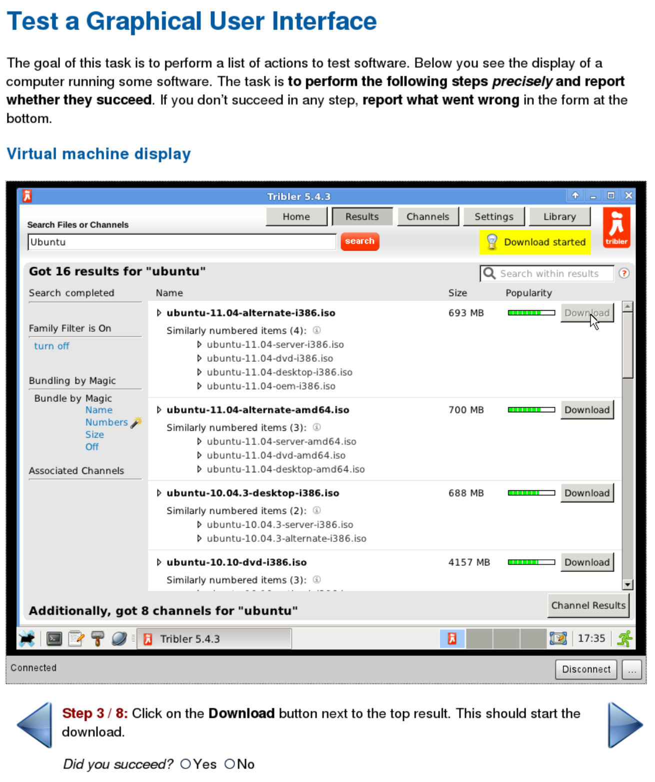
⑤ — When the worker clicks the submit button, the answers to each step are sent to MTurk.

⑥ — The VM server periodically fetches assignment results from MTurk. When a worker's assignment result is received, the acceptance check script is run inside the VM of the worker, which is then terminated.

⑦ — Developers can browse HITs and submitted assignments on a website served by the VM server. Figure 4 shows an example of an assignment result page. In this example taken from an actual HIT[6], the worker reported that he or she was unable to start the download initially. The recording of the VM's display, which can be played back by clicking on the "Video" link, allowed the Tribler developers to see the problem immediately: after clicking on "Download", Tribler asynchronously fetches the corresponding Torrent file; under some conditions, however, it would not actually start the download after obtaining this file.

HITs can be created on an *ad hoc* basis, e.g., for usability testing experiments. However, most HITs are created automatically from a continuous build system. For instance, for every commit in Tribler's version control system, the task in Figure 2 is instantiated with the latest checkout.

[6]http://nixos.org/mturk/job/6



Test a Graphical User Interface

The goal of this task is to perform a list of actions to test software. Below you see the display of a computer running some software. The task is **to perform the following steps *precisely* and report whether they succeed**. If you don't succeed in any step, **report what went wrong** in the form at the bottom.

Virtual machine display

Tribler 5.4.3

Search Files or Channels: Ubuntu

Got 16 results for "ubuntu"

Search completed	Name	Size	Popularity	
Family Filter is On	**ubuntu-11.04-alternate-i386.iso**	693 MB		Download
turn off	Similarly numbered items (4): ubuntu-11.04-server-i386.iso, ubuntu-11.04-dvd-i386.iso, ubuntu-11.04-desktop-i386.iso, ubuntu-11.04-oem-i386.iso			
Bundling by Magic	**ubuntu-11.04-alternate-amd64.iso**	700 MB		Download
Bundle by Magic: Name, Numbers, Size, Off	Similarly numbered items (3): ubuntu-11.04-server-amd64.iso, ubuntu-11.04-dvd-amd64.iso, ubuntu-11.04-desktop-amd64.iso			
Associated Channels	**ubuntu-10.04.3-desktop-i386.iso**	688 MB		Download
	Similarly numbered items (2): ubuntu-10.04.3-server-i386.iso, ubuntu-10.04.3-alternate-i386.iso			
	ubuntu-10.10-dvd-i386.iso	4157 MB		Download
	Similarly numbered items (3):			

Additionally, got 8 channels for "ubuntu"

Step 3 / 8: Click on the **Download** button next to the top result. This should start the download.

Did you succeed? ○Yes ○No

Figure 3. A Tribler GUI testing HIT as it appears in a worker's web browser, at step 3 of the task

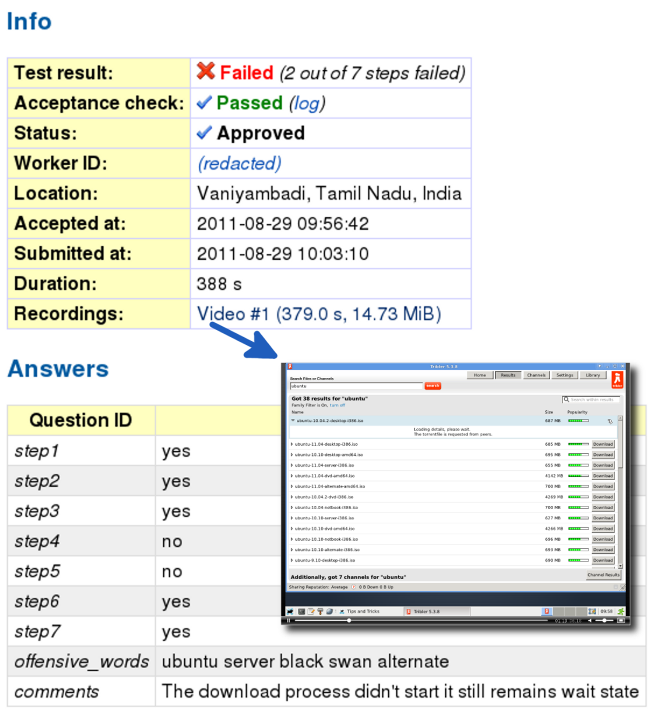
Our prototype system considers a GUI test HIT to *pass* if at least 60% of all assignments indicated no problems (that is, answered "yes" to all questions) and passed the automatic acceptance check. It is considered to *fail* if at least 60% of all assignments indicate a problem (i.e., answered "no" to at least one question). Otherwise, the result is *inconclusive*. The result is often useful and revealing to developers in any of these cases, since even a passing or inconclusive test can show interesting interaction patterns.

IV. Implementation

We now describe several significant technical aspects of our prototype system.

A. Building VMs

To specify and instantiate VMs, we applied our previous work on automating system tests [19]. There, we used declarative models such as the one shown in Figure 5 (explained below) to build virtual machines in which to run automated system tests. Such specifications consist of a description of the desired configuration of each machine, along with an imperative script that runs test actions on the machines. This approach builds on NixOS [20], a Linux distribution based on the purely functional Nix package manager [21], to ensure that VMs can be instantiated efficiently (i.e., without building large disk images), an important property for use in continuous build systems. Here, instead of doing automated tests, we use this method to prepare virtual machines for interactive tests.



Info

Test result:	✗ Failed *(2 out of 7 steps failed)*
Acceptance check:	✓ Passed *(log)*
Status:	✓ **Approved**
Worker ID:	*(redacted)*
Location:	Vaniyambadi, Tamil Nadu, India
Accepted at:	2011-08-29 09:56:42
Submitted at:	2011-08-29 10:03:10
Duration:	388 s
Recordings:	Video #1 (379.0 s, 14.73 MiB)

Answers

Question ID	
step1	yes
step2	yes
step3	yes
step4	no
step5	no
step6	yes
step7	yes
offensive_words	ubuntu server black swan alternate
comments	The download process didn't start it still remains wait state

Figure 4. An assignment result page, showing a failing test

Figure 5 shows the specification of the virtual machine used by the Tribler GUI test in Figure 2. This is a *function* that takes as an argument the path to the source code of Tribler (at point ①). This enables it to be called from a continuous build system with the latest revision as an argument to build the corresponding VM and HIT. The function returns a call to makeMTurkTest at ② that causes the Nix package manager to build a script that starts the virtual machine. All dependencies of this script – e.g., the Linux kernel, the X11 window server, the Xfce desktop environment and Tribler – are built as well if necessary. (Nix can be thought of as a high-level, purely functional Make that works at the level of packages.)

The attribute machine at ③ defines the configuration of the virtual machine. For instance, it states that the VM should have 1 GiB of RAM. For convenience, it is possible to factor out commonality in VM configurations into separate modules: thus, the Tribler configuration imports at ⑤ several modules that set up the Xfce desktop environment, and so on. The VM also includes a Tribler package built from source code at ④. (The function mkDerivation builds packages from source.)

A Perl script to bring the VM into the desired state for the GUI test is defined in the attribute prepareVM at ⑥. After implicitly starting the VM, it executes actions such as waiting until the Xfce desktop environment has finished booting (as indicated by the appearance of the xfce4-panel

```
{ triblerSrc }: ① makeMTurkTest { ②

  machine = ③
    let tribler = stdenv.mkDerivation
          { ... src = triblerSrc; ... }; ④
    in
    { require = [ ./common-xfce.nix ... ]; ⑤
      environment.systemPackages = [ tribler ];
      virtualisation.memorySize = 1024;
    };

  prepareVM = '' ⑥
    # Wait for the Xfce desktop to start.
    $machine→waitForWindow(qr/xfce4-panel/);
    # Start Tribler, wait for it to appear.
    $machine→execute("su - user -c
      'DISPLAY=:0.0 tribler &'");
    $machine→waitForWindow(qr/Tribler/);
    # Wait for Tribler to gather some buzz.
    $machine→sleep(120);
  '';

  acceptanceCheck = '' ⑦
    $machine→succeed('[ -n "$(find
      /home/user/Desktop/TriblerDownloads
      -maxdepth 1 -type f)" ]');
  '';
}
```

Figure 5. Virtual machine specification for the Tribler task

window), starting Tribler, and sleeping for a while to allow Tribler to connect to peers on the Internet so that the Torrent search facility works.

Likewise, the attribute acceptanceCheck at ⑦ defines a script that checks whether the worker performed the test successfully. (If it is omitted, assignment results are unconditionally accepted.) Here, it tests whether a file has appeared in the TriblerDownloads directory. This indicates that Tribler has started to download at least one file.

B. Running VMs

For each HIT, the VM server pre-starts a pool of virtual machines. It is necessary to start VMs in advance to ensure that when a worker connects, he can access a VM immediately, rather than having to wait for a new VM to boot.

Virtual machines are executed using QEMU/KVM, a virtualisation system for Linux. We enabled KVM's *same-page merging* feature, a memory deduplication method that allows identical memory pages in different VMs to be merged into a single page in the host's physical RAM [22]. Since the VMs in a pool are nearly identical, this allows significantly more VMs to run on a host. For instance, during the second A/B test described in Section V, it cut memory consumption for a pool of 20–25 VMs from about 6.6 GiB to 3.6 GiB.

C. Accessing VMs

The HTML page for the HIT contains a Flash control that uses the VNC remote desktop protocol to allow access to the virtual machine. QEMU/KVM provides a built-in VNC server to allow clients to access the VM remotely. To access it, the client connects to a *VNC multiplexer* that selects the correct VM instance for the client (based on its HIT and assignment IDs), then proxies the connection onward to the VNC server of the selected VM instance. The VNC multiplexer is also responsible for producing a video recording of the session.

The first time that a worker connects with a given $\langle$hitId, assignmentId$\rangle$ tuple, the VNC multiplexer picks a VM from the associated HIT's pool of unused VMs. At this time, a new VM is started to maintain the size of the pool of unused VMs. The selected VM is thereafter considered *in use* and is persistently associated with that assignment. Thus, if the worker connects again with the same tuple (e.g., after restarting his browser), the same VM will be selected.

We log when the worker selects an answer to a question (i.e., clicks on the Yes or No radio buttons in Figure 3). This is useful because it allows viewers to jump directly to the point in the recording corresponding with a step in the task.

To obtain the results of submitted assignments, the VM server periodically polls MTurk. When an assignment result is received, the associated VM's acceptance check is executed, and the VM is terminated. When the target number of assignments for a HIT has been submitted (e.g., 10 for the Tribler test in Figure 2), all VMs associated with the HIT are terminated. This includes unused VMs and VMs associated with abandoned assignments.

V. Evaluation

We have performed a number of experiments to determine the feasibility of crowdsourcing of GUI tests. Specifically, we set out to answer the following questions:

RQ1 Are workers *technically* able to perform the tasks? For instance, if most potential workers have very slow Internet connections, or if the latency is very high, this may make it impossible in practice to crowdsource GUI tests.

RQ2 Is crowdsourcing a feasible approach for continuous testing? This requires that sufficiently many workers correctly determine whether a test passes or fails.

RQ3 How long do crowdsourced GUI tests take, i.e., what is the average runtime of a HIT?

RQ4 Is crowdsourcing a feasible approach for usability experiments? In particular, in such experiments it is often necessary to measure task completion times. If these are extremely random due to factors such as worker network latency and bandwidth, it may be infeasible to get statistically significant results.

RQ5 How do we design a HIT so that HIT execution time is minimised?

We do not directly evaluate economic usefulness (that is, whether crowdsourcing is actually cheaper than conventional testing); we do touch on this in the next section.

Country	Workers	Assign-ments	Median speed (KiB/s)	Mean ping (ms)
India	247	490	33.7	329
USA	42	49	200.3	202
UK	11	28	535.1	52
Pakistan	8	9	24.6	299
Romania	7	14	468.0	25
(27 countries omitted)				
Total	398	700	48.0	260

Table I
WORKER DISTRIBUTION BY COUNTRY

A. Experimental Setup

All experiments (i.e., all virtual machines) were run on a single Dell PowerEdge R815 machine with 4 12-core AMD 6164 HE CPUs and 96 GiB of RAM. This server is located in the Netherlands, which is, as we shall see, far from the majority of workers, thus affecting latency negatively.

The workers participating in our HITs were self-selected; we had no control over which workers accepted a HIT. The Mechanical Turk allows requesters to require that workers meet certain qualifications, such as geographical location, a minimum acceptance rate for past assignments, or passing a *qualification HIT* (a HIT that has to be performed before the worker can do other HITs). In order not to introduce any bias into the set of workers, we did not require qualifications from workers.

B. Worker Demographics

In our experiments, we gathered various bits of information about workers that are relevant to HIT design and technical and economic feasibility. These include information about the location of the workers, their network bandwidth and latency, and their display resolutions.

In the evaluation below, we ran 51 HITs on the MTurk. In total, 398 unique workers from 32 different countries submitted 700 assignments. Table I shows the top 5 worker countries, along with the median download speed and average ping time between the worker and the VM server. To estimate a worker's download speed, we instrumented our HITs with some JavaScript to have the worker's browser fetch a file from the VM server. Ping time was determined by having the VNC multiplexer perform a ping to the client when it connects.

As the table reveals, the vast majority of workers come from India. This is unsurprising, since India is the only country besides the U.S. where Amazon pays workers directly; in all other countries, workers receive Amazon store credits. Connection speeds of Indian workers are fairly low. This has an effect on task completion time, as we shall see below, but not a fatal one. Some workers complained that access to the VM was slow, but in most cases did manage to complete the task. (Our HIT summary did advice workers that a "reasonably" fast Internet connection was recommended.)

Addressing **RQ5**, we also instrumented HITs to log the screen resolution of workers. This is a particularly important data point, because a good work flow is only possible if the VM display and the current step in the task are simultaneously visible (as in Figure 3). If the VM screen is too large, workers will have to scroll or pan frequently, significantly increasing the task completion time. We found that the most common resolutions are fairly low: 25.3% of workers have a 1024x768 screen, 20.7% have 1366x768 and 11.8% have 1280x800. We initially used a VM resolution of 1024x768, but after this analysis, we lowered the resolution to 800x600 (for most HITs) and 640x480 (for some HITs).

C. Continuous Testing

To address **RQ1** and **RQ2**, we created a number of GUI testing task descriptions and attached them to a continuous build system. That is, HITs were instantiated when developers committed changes, subject to a minimum time interval between HITs. For almost all HITs, we requested 10 assignments to be submitted. The tests are the following:

- *Tribler test*: the test shown in Figures 2 and 5. Tribler is written in Python and uses the wxPython GUI toolkit. The HIT is built from Tribler's Subversion repository.
- *KDE login/logout test*: KDE is a desktop environment for Unix. The VM preparation script boots KDM, the KDE login manager. The test asks workers to login, start the Konqueror web browser and visit a given URL, then logout. KDE is written in C++ and uses the Qt GUI toolkit. This and the remaining tests are built from the repository of the NixOS Linux distribution; thus they are continuous system tests of NixOS, rather than KDE.
- *KDE USB stick mounting test*: The preparation script provides a logged-in KDE session. The test is to open the Dolphin file manager, click on a USB stick to mount it, copy a file from the USB stick, unmount the USB stick, and open the copied file. (This test uses QEMU's ability to virtualise USB hardware; the "USB stick" is a disk image passed to QEMU.) This is a good system test because the ability to mount external devices in KDE depends on many system and desktop components working in concert (e.g., udev, udisks, PolicyKit, and ConsoleKit).
- *Xfce editor test*: Xfce is another desktop environment for Unix. The test is to create a file in the Xfce editor application, save it, then reopen it in the Xfce file manager. Xfce is written in C and uses the GTK+ toolkit.

Table II summarises the results of the HITs instantiated from these tests[7]. For each test, we list the reward per assignment; the number of HITs created; the average runtime of the HITs, that is, the time between creation of the HIT and submission of the last assignment; the number of assignments submitted or abandoned; the median duration

[7]Detailed results of these HITs, including all video recordings, are available at http://nixos.org/mturk.

	Tribler	KDE login	KDE mount	Xfce
Reward	$0.15	$0.10	$0.10	$0.10
# Hits	14	10	11	10
Average runtime	2.0 h	3.6 h	2.0 h	2.1 h
# Submitted	145	100	115	100
# Abandoned	9	9	11	7
# Workers	112	86	94	85
Median duration	314.0 s	327.5 s	240.0 s	246.5 s
Hourly rate	$1.72	$1.10	$1.50	$1.46
% Correct	66.9%	77.0%	68.7%	82.0%
% Tech. issues	5.5%	6.0%	5.2%	3.0%
% Misunderstood	2.1%	6.0%	13.9%	2.0%
% Fraud	3.4%	4.0%	2.6%	7.0%

Table II
RESULTS OF THE CONTINUOUS TESTING HITS

of assignments, that is, the time between HIT acceptance and result submission; the effective hourly rate paid to the median worker; the percentage of submissions that correctly classified the HIT as *pass* or *fail*, as appropriate; and the number of assignments where the worker reported running into technical issues (e.g., the VNC Flash control gave an error), where the worker misunderstood the task, and where the worker submitted a fraudulent result (e.g., clicked "yes" on all steps, while the recording shows no activity). This classification was done manually by viewing the recordings.

We manually injected faults into the systems under test in some HITs. For instance, for the KDE USB mounting test, we created a HIT with a broken PolicyKit configuration, preventing users from mounting external devices.

Gratifyingly, some reported failures were the results of actual bugs introduced (unwittingly) by NixOS developers. For instance, in one case[8], an upgrade from GTK+ 2.24.5 to 2.24.6 caused a regression in the Xfce editor: the *Save as* dialog box was suddenly much larger, causing the *Save* button to fall off the screen. Some workers worked around this issue by moving the window and reported no error; others flagged failure and reported in the comment field that there was no *Save* button Note that this kind of bug might not be found by many automated testing frameworks, because the dialog responds fine to simulated abstract events. (This probably counts as a "surprise" in the sense of [8].)

As Table II shows, the number of incorrect assignment results is fairly high (though at 10 assignments per HIT, the correct results generally outvote the incorrect ones). We analysed the assignments to see why this was the case. One important cause is that workers are often sloppy in performing steps precisely: for instance, in the Xfce test, users are asked to create a file named test.txt; however, many workers used a different file name (e.g., bla.txt), causing the acceptance check to fail. For the Tribler test, an interesting cause is visual lag: many workers reported that the third step (clicking on "Download") did nothing. This is likely because the briefly flashing "Download started" message is not seen on slow connections.

[8]http://nixos.org/mturk/job/28

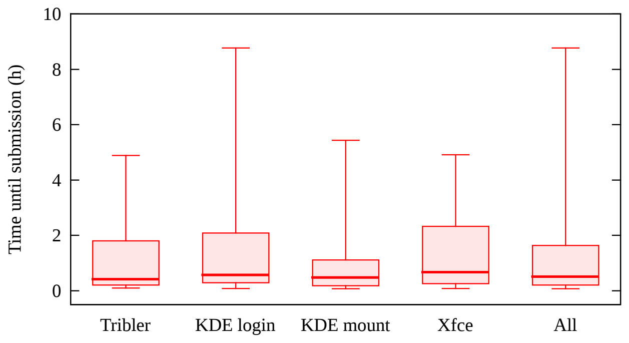


Figure 6. Box plots showing the distribution of assignment submission times relative to the creation time of the HIT

We measured how long it takes for results to come in (**RQ3**). The box plots in Figure 6 show, for each of the continuous tests and all together, how long after the creation of the HIT assignments were submitted. (The boxes denote the upper and lower quartile; the line in each box is the median; and the whiskers denote the extremes.) This shows that most submissions come in quickly, but the outliers cause some HITs to take several hours to complete.

D. Usability Testing

To discover whether crowdsourcing is a feasible method for GUI usability experiments (**RQ4**), we ran two *A/B tests* [23] to compare user performance between different variants of Tribler. Specifically, we were interested in evaluating the usefulness of an experimental Tribler feature called *bundling*, which groups related search results together on the basis of a number of criteria, such as file name or size. For instance, episodes of a television series may be grouped together. Thus, the main A/B experiment is to create a HIT in which half the users get a Tribler instance with bundling disabled, and the other half get one with bundling enabled. This is implemented by filling the VM pool for a HIT with two kinds of VMs (which can be arbitrarily different). A worker is thus randomly assigned either the A or B variant by the VNC multiplexer. The goal of the experiment is to establish whether there is a statistically significant difference in the average time between a user initiating a search, and pressing the download button on the appropriate search result. That is, the null hypothesis H_0 is that $\mu_A = \mu_B$ (where μ is the mean time interval), while the alternative hypothesis H_1 is that $\mu_A < \mu_B$.

However, we worried about the main threat to the validity of results from a crowdsourced experiment: the large variance in worker connection latency and bandwidth (as seen above), which in turn can cause a large variance in assignment completion times. Thus, to evaluate the experimental method itself, we first performed an A/B test between the non-bundling variant, and the non-bundling variant *with an*

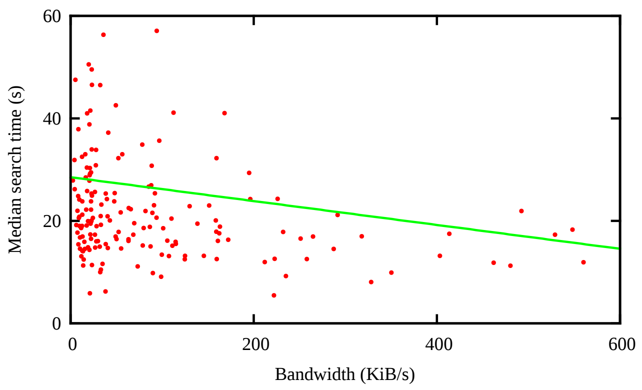


Figure 7. Worker bandwidth vs. median search time in the Tribler A/B tests, with best-fit linear regression line

artificial delay of 2 seconds in the presentation of search results to the user. We then expected to observe a difference in the average time interval (ideally, +2 seconds).

We instrumented Tribler to log search and download actions, and created a HIT that asked users to enter a variety of search terms and find a specific file in the resulting list. The HIT requested workers to perform the task without interruptions. We ran the HIT with 100 assignments (and thus 100 workers), which took 28 h 58 m to complete. With a reward of $0.25 per assignment, this experiment cost $25. This yielded 354 measurements for the A (non-bundling) variant and 330 for the B (delayed) variant. The median interval was 19.6 s for A and 21.7 s for B, neatly conforming to the 2 second delay in B. The arithmetic means, however, were 30.8 s versus 28.9 s, mostly because the A set had a few extreme outliers. The trimmed means obtained by discarding the 10% highest measurements were 21.3 s versus 22.2 s. Discarding the 25% highest measurements to account for the skew in the distribution, Student's t-test rejects H_0 at $P = 0.049$, a significant result.

With the same setup, we performed an A/B test comparing the non-bundling and bundling variants. This HIT took 28 h 48 m to complete and yielded 332 measurements for the non-bundling variant and 269 for the bundling variant. Here there was no clear difference between the variants: the trimmed means were 19.9 s and 21.4 s, and the medians 18.3 s and 19.2 s, respectively. The null hypothesis was *not* rejected at $P = 0.494$. Thus the experiment suggests that bundling does not lead to faster search result interpretation by users.

Returning to **RQ1**, Figure 7 shows the relationship between a worker's bandwidth and the median search time for the queries performed by each worker during the A/B tests. This suggests that faster connections do reduce task completion time, but the effect is fairly modest; slow connections are no major impediment.

E. Summary of Results

We now summarise the conclusions to the research questions. We can answer **RQ1**, whether workers are technically able to perform the tasks, in the affirmative. **RQ2** asks whether crowdsourcing is feasible for continuous testing; this is definitely the case for simple tasks, but for complex tasks, more work on task design, worker qualification and result processing is needed. **RQ3** concerns the runtime of GUI testing HITs, which we have shown to be on the order of a few hours. Our Tribler A/B tests show that the answer to **RQ4** – whether crowdsourcing can be employed for usability experiments – is positive. **RQ5** is an open-ended question regarding HIT design; our contribution is that VM resolutions should be minimised and task steps should be simple and unambiguous.

F. Threats to Validity

A threat to internal validity is that our evaluation was conducted over a limited amount of time (around a month): it is possible, for instance, that workers performed our HITs because of their novelty and may lose interest over time. This could cause HIT runtime to go up in the future.

There are a number of threats to external validity – the extent to which our results can be generalised to other situations. The most important is that crowdsourcing assumes that there is a sufficiently large pool of people motivated and able to work on the task. If the motivation is financial, the crowdsourcing "business model" depends in part on the existence of countries with low per-capita GDP, good Internet connectivity, and a large population that understands the language of the task. If these are not available, crowdsourcing may not work, or may be more expensive. In the latter case, crowdsourcing is still useful for its elasticity (the ability to quickly attract more workers).

Second, for usability tests, we assume that the worker possesses the requisite knowledge to work with the application. This is the case for applications that target a general audience, such as Tribler, but may not hold for specialised applications. For instance, we cannot use arbitrary workers in a usability study of an Eclipse plugin. A qualification HIT could address this, but there simply may not be enough skilled and interested workers on MTurk.

Third, the task formalism assumes that tasks can easily be described in words (e.g., "click on button Y"). For some types of interaction, this may be insufficient. Consider a drawing program where we want the worker to draw and manipulate shapes in a certain way; it may be too difficult to convey the desired motions to the worker. However, one can imagine HITs than contain screenshots or recordings of a reference session that the worker is asked to replicate.

The continuous tests in Section V were fairly unsystematic, being mostly exploratory in nature: we did not attempt to ensure sufficient coverage of the systems under test. This leads to an important economic consideration that we have neglected here: how many HITs are necessary, and at what (total) price, to provide sufficient test coverage for a given system? To discover this, it would be interesting to use crowdsourcing in conjunction with a more systematic GUI testing method (e.g. [8]).

VI. Conclusion

In this paper, we have described a method for crowdsourcing of GUI tests based on instantiating the system under test in virtual machines that are served to a geographically dispersed pool of workers. We conclude that this approach works well in terms of technical feasibility: while many workers have slow connections, this does not prevent them from completing tasks successfully.

For continuous testing, our experiments show that crowdsourcing is a very promising approach, even though the number of incorrect results is somewhat high. We believe that better HIT design and worker qualification can improve this in the future. For usability studies, our experiments demonstrate that crowdsourcing enables a much larger group of participants to be mobilised at much lower cost than would be feasible in a conventional approach.

Acknowledgments: We wish to thank Martha Larson for her advice on crowdsourcing and her comments on the design of the A/B test; Niels Zeilemaker for discussions and fixing bugs in Tribler; and of course all workers who participated in our HITs. This research was supported by NWO-JACQUARD project 638.001.208 (*PDS: Pull Deployment of Services*) and the NIRICT LaQuSo Build Farm project.

References

[1] A. M. Memon, "A comprehensive framework for testing graphical user interfaces," Ph.D. dissertation, University of Pittsburgh, 2001.

[2] X. Yuan and A. M. Memon, "Using GUI run-time state as feedback to generate test cases," in *29th Intl. Conf. on Software Engineering (ICSE '07)*, May 2007, pp. 396–405.

[3] Q. Xie, "Developing cost-effective model-based techniques for GUI testing," in *28th Intl. Conf. on Software Engineering (ICSE '06)*. ACM, 2006, pp. 997–1000.

[4] Q. Xie and A. M. Memon, "Designing and comparing automated test oracles for GUI-based software applications," *ACM Transactions on Software Engineering and Methodoly*, vol. 16, Feb. 2007.

[5] A. M. Memon and Q. Xie, "Using transient/persistent errors to develop automated test oracles for event-driven software," in *19th IEEE Intl. Conf. on Automated Software Engineering (ASE '04)*. IEEE Computer Society, 2004, pp. 186–195.

[6] A. M. Memon, I. Banerjee, and A. Nagarajan, "What test oracle should I use for effective GUI testing?" in *IEEE Intl. Conf. on Automated Software Engineering (ASE '03)*. IEEE Computer Society, Oct. 2003, pp. 164–173.

[7] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs," in *8th ACM SIGSOFT Intl. Symp. on Foundations of Software Engineering (FSE-8)*. New York, NY, USA: ACM, 2000, pp. 30–39.

[8] L. White and H. Almezen, "Generating test cases for GUI responsibilities using complete interaction sequences," in *11th Intl. Symp. on Software Reliability Engineering (ISSRE '00)*. IEEE Computer Society, 2000, pp. 110–121.

[9] B. A. Myers, "State of the art in user interface software tools," in *Advances in Human-Computer Interaction*, H. R. Hartson and D. Hix, Eds., 1993, vol. 4, pp. 110–150.

[10] J. Howe, "The rise of crowdsourcing," *Wired*, vol. 14, no. 6, Jun. 2006.

[11] P. G. Ipeirotis, "Mechanical Turk: Now with 40.92% spam," http://www.behind-the-enemy-lines.com/2010/12/mechanical-turk-now-with-4092-spam.html, Dec. 2010.

[12] V. Ambati, S. Vogel, and J. Carbonell, "Active learning and crowd-sourcing for machine translation," in *7th Conf. on International Language Resources and Evaluation (LREC'10)*. Valletta, Malta: European Language Resources Association (ELRA), May 2010.

[13] O. F. Zaidan and C. Callison-Burch, "Crowdsourcing translation: Professional quality from non-professionals," in *49th Annual Meeting of the Assoc. for Computational Linguistics: Human Language Technologies*, Jun. 2011, pp. 1220–1229.

[14] S. Novotney and C. Callison-Burch, "Cheap, fast and good enough: automatic speech recognition with non-expert transcription," in *Human Language Technologies: 11th Annual Conf. of the North American Chapter of the Assoc. for Computational Linguistics*, 2010, pp. 207–215.

[15] G. Paolacci, J. Chandler, and P. G. Ipeirotis, "Running experiments on Amazon Mechanical Turk," *Judgment and Decision Making*, vol. 5, no. 5, pp. 411–419, Aug. 2010.

[16] M. Buhrmester, T. Kwang, and S. D. Gosling, "Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality, data?" *Perspectives on Psychological Science*, vol. 6, no. 1, Jan. 2011.

[17] E. Adar, "Why I hate Mechanical Turk research (and workshops)," in *CHI 2011 Workshop on Crowdsourcing and Human Computation*, May 2011.

[18] J. A. Pouwelse, P. Garbacki, J. Wang, A. Bakker, J. Yang, A. Iosup, D. H. J. Epema, M. Reinders, M. R. van Steen, and H. J. Sips, "Tribler: a social-based peer-to-peer system," *Concurrency and Computation: Practice and Experience*, vol. 20, pp. 127–138, Feb. 2008.

[19] S. van der Burg and E. Dolstra, "Automating system tests using declarative virtual machines," in *21st IEEE Intl. Symp. on Software Reliability Engineering (ISSRE '10)*. IEEE Computer Society, Nov. 2010.

[20] E. Dolstra and A. Löh, "NixOS: A purely functional Linux distribution," in *13th ACM SIGPLAN Intl. Conf. on Functional Programming (ICFP 2008)*. ACM, Sep. 2008.

[21] E. Dolstra, E. Visser, and M. de Jonge, "Imposing a memory management discipline on software deployment," in *26th Intl. Conf. on Software Engineering (ICSE 2004)*. IEEE Computer Society, May 2004, pp. 583–592.

[22] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Linux Symposium*, Jul. 2009.

[23] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: survey and practical guide," *Data Mining and Knowledge Discovery*, vol. 18, pp. 140–181, Feb. 2009.