Tsumiki: A Meta-Platform for Building Your Own Testbed

Justin Cappos\textsuperscript{ny} \hspace{1em} Yanyan Zhuang\textsuperscript{co} \hspace{1em} Albert Rafetseder\textsuperscript{ny} \hspace{1em} Ivan Beschastnikh\textsuperscript{•} \\
\textsuperscript{ny} New York University \hspace{1em} \textsuperscript{co} University of Colorado, Colorado Springs \hspace{1em} \textsuperscript{•} University of British Columbia

Abstract—Network testbeds are essential research tools that have been responsible for valuable network measurements and major advances in distributed systems research. However, no single testbed can satisfy the requirements of every research project, prompting continual efforts to develop new testbeds. The common practice is to re-implement functionality anew for each testbed. This work introduces a set of ready-to-use software components and interfaces called Tsumiki to help researchers to rapidly prototype new testbeds without substantial effort. We derive Tsumiki’s design using a set of component and interface design principles, and demonstrate that Tsumiki can be used to implement new, diverse, and useful testbeds. We detail a few such testbeds: a testbed composed of Android devices, a testbed that uses Docker for sandboxing, and a testbed that shares computation and storage resources among Facebook friends. A user study demonstrated that students with no prior experience with networked testbeds were able to use Tsumiki to create a testbed with new functionality and run an experiment on this testbed in under an hour. Furthermore, Tsumiki has been used in production in multiple testbeds, resulting in installations on tens of thousands of devices and use by thousands of researchers.

Index Terms—Networked testbeds, distributed systems

1 INTRODUCTION

Testbeds—such as RON [1], PlanetLab [2], Emulab [3], and GENI [4]—play an important role in evaluating network research ideas. These testbeds enable researchers to run software on thousands of devices, and have significantly improved the validation of networking research. For example, RON and PlanetLab have been extensively used to evaluate early research on Distributed Hash Table (DHT), while BLSmark [5] and SamKnows [6, 7] have impacted Internet policy decisions at the Federal Communications Commission (FCC).

Since each testbed has unique capabilities and each project has different requirements, no single networked testbed fits all needs. For example, PlanetLab is well suited for hosting long-running Internet services across a wide area. However, PlanetLab gives a skewed view of Internet connectivity [8, 9, 10]. This deficiency prompted researchers to create new testbeds, such as SatelliteLab [11], BLSmark [5, 7], and Dasu [12]. These testbeds support measurements from end hosts or edge networks and capture more network and geographic diversity than PlanetLab. We expect that new testbeds will continue to be developed as new technologies emerge.

Today it is common practice to build a new testbed without reusing software from existing testbeds. This is because existing testbeds are often customized to a particular environment and use case. For example, a researcher may want to reuse the PlanetLab software to build a testbed for running experiments on Android devices. However, the PlanetLab node software requires a custom Linux kernel with special virtualization support. These customizations are non-trivial to port to mobile hardware. Further, many of the PlanetLab abstractions, such as the “site” notion, do not readily apply. Hence, the conventional wisdom is that it is simpler to develop a new testbed software stack from scratch, instead of building on existing software.

In this work we show that code reuse between testbeds does not limit the construction of diverse testbeds. We formulate a set of testbed design principles and use these, in concert with existing designs, to propose a component-based model that we call Tsumiki. Tsumiki is a meta-platform for testbed construction that makes it easy to build and customize new units from existing components.

Tsumiki consists of seven software components that are configurable and replaceable. Tsumiki components are characterized by open, well-defined APIs that enable different component realizations (i.e., diverse implementations of a component using the same interfaces), to work together without strong dependencies. Although the interfaces are tightly specified, there can be many realizations of each component, and any realization can be customized to a particular environment or testbed feature. For example, the sandbox component can be realized as a Docker container (Section 5), or a programming language sandbox (Section 5), etc. The experiment manager component can be realized as a tool for automated deployment of long-running experiments, a parallel shell, or a GUI-based tool.

We used the Tsumiki principles in 2008 to design and build Seattle [13]. Seattle allows research experimentation on end-user devices, which include tens of thousands of smartphones, laptops and desktops. It is used in academic research and pedagogy in dozens of universities and institutions. Today, Seattle serves testbed software updates to over 40K devices. Seattle, and several other subsequent Tsumiki testbeds, were built as community efforts, with

1. The word tsumiki means “building blocks” in Japanese.
The primary scientific contribution of this work are four principles and our application of these principles to derive a modular testbed design called Tsumiki. This design introduces a set of components that can be used by testbed developers to extend, replace, or omit components in existing testbeds, while allowing them to reuse functionalities.

In our evaluation we demonstrate three benefits of the Tsumiki design:

**Benefit 1: Tsumiki can be used to construct a variety of testbeds.** Tsumiki has been used to build and deploy testbeds like Tomato (a network virtualization and emulation testbed), Sensibility (a testbed that provides IRB-approved access to sensors on Android devices), and Seattle (a testbed for networking, security, and distributed systems research and education). Section 6 describes Ducky, a new testbed which uses Docker to sandbox experiment code. In Section 7 we also describe Social Compute Cloud, a testbed that was built by an external group. It uses social networks to locate computational and storage resources. In Section 8 we present other production testbeds that use Tsumiki.

**Benefit 2: Developing testbeds with Tsumiki is easy.** It took one of the authors three hours to build the Ducky testbed. We also report on a user study in which participants with little experience in networked testbeds used Tsumiki to create a custom testbed, and deploy an experiment, all in under an hour (Section 8).

**Benefit 3: Tsumiki enables researchers to construct testbeds that are useful for research.** In Section 9 we show how the Sensibility Testbed can be used to reproduce a study of WiFi rate adaptation algorithms. This study helped us uncover a mistake in Ravindranath et al.’s description of the algorithm. Over the past eight years, dozens of projects have also used Tsumiki-based testbeds [14], [18], [19], [20], [21], [22], [23].

The remainder of this paper is structured as follows: Section 2 states the goals and non-goals guiding the design of Tsumiki. Section 3 derives Tsumiki’s main design principles. Section 4 discusses the components and interfaces that follow from Tsumiki’s principles. Sections 5, 6, and 9 present five practical testbed implementations based on Tsumiki. In Section 8 we show a study in which students with little prior knowledge in testbeds successfully use Tsumiki to construct a testbed with new functionalities. Section 10 discusses the limitations that Tsumiki is subject to. In Section 11 we review related work in the design space, and Section 12 concludes.

## 2 Goals, non-goals, and definitions

**Goals.** The goals of this work are to: (1) reduce the development time and effort in building testbeds, (2) improve the compatibility of testbed components so they can be used in diverse testbeds, and (3) generalize and diversify existing testbed and component designs to extend the range of supported use cases.

**Non-goals.** There are also several important non-goals. Our goal is not to specify a canonical set of components that all testbeds must include. Also, we are not aiming to have the eventual components provide optimal performance, memory footprint, etc. Much like the tradeoffs that arise from using a high level programming language, Tsumiki decreases development time but may introduce a penalty over an optimized testbed design. Furthermore, Tsumiki does not attempt to eliminate the possibility of testbed misconfiguration or make it impossible to implement buggy components. However, Section 6 demonstrates that it is easy for students who have no experience with testbeds to quickly create new testbeds that function correctly.

**Definitions.** This paper refers to a *component* when discussing a basic unit of modularity within a testbed. A component is a unit of composition with clearly specified interfaces and explicit context dependencies [24]. Each component has different *realizations*, or implementations that work in different ways. Valid component realizations conform to the same precisely defined interfaces for inter-component communication. However, they may differ in other ways, such as the user interface or the included optimizations.

Throughout the paper we use the following testbed-related terminology. *Devices* are hardware resources that host experiments and are owned by *device owners*. An *experimenter* is a person who runs experiments on some set of devices. A *testbed infrastructure* includes some testbed hardware and testbed software. The testbed hardware is usually provided and operated by a testbed *provider*. The testbed software is a set of services that manage the operation of the testbed, such as software to track devices. The testbed software is created by *testbed developers*. While the testbed developers, testbed provider, experimenters, and device owners may be the same group of people, this is not always the case.

## 3 Guiding design principles

The Tsumiki model was developed using a set of guiding principles. The first two principles are related to selection and lay-out of the testbed components.

**P1 Principle of least trust.** Whenever a component would require a trust relationship between two parties that may not trust each other, we separate the component into two components along the trust boundary.

**P2 Design for omission principle.** The design should allow components to be omitted if their functionality is not necessary for a specific testbed.

P1 makes trust relationships between components explicit. This allows experimenters, providers, and developers to select or construct component realizations independently.

Not all testbeds require all components. P2 guides Tsumiki’s design to support testbeds that are designed to offer only the features needed to perform effectively.

The next two principles relate to component interfaces.

**P3 Non-extensible inter-component interfaces with opaque parameters.** Inter-component interfaces must not change to provide a guarantee of backward compatibility. To be flexible in the otherwise rigid interface, we use opaque parameters.

**P4 Flexible human-visible interfaces.** Interfaces that are visible to experimenters or device owners must be flexible and allowed to vary between different realizations.
Device owner

Experiment manager

Testbed infrastructure

locus service

Device host system
device manager

sandbox

resource manager

installer builder

lookup service

Experimenter’s local host
device manager

Fig. 1: Tsumiki components (boxes) and interfaces (arrows, pointing in the direction of communication). Shaded components have interfaces used by a human.

We introduce P3 to support our goal of compatibility between testbed components and interoperability between testbeds. P3 requires that inter-component interfaces, and the resulting inter-component interactions, be precisely defined. This allows testbed developers to make assumptions about component behavior even if they do not control its final realizations. To reconcile this principle with the goal of supporting a variety of testbeds, we frequently use opaque parameter values for inter-component interfaces. These parameter values must be acted on in only specific and precisely defined ways.

P4 relates to interfaces that a human would use to interact with a component, such as a UI or a programming language in a sandbox. P4 allows a component to define this interface in the way that best fits its anticipated use. In contrast to the inter-component interfaces, we do not define these interfaces. As a result, such human-visible interfaces can change to meet the requirements of a new testbed.

4 Tsumiki’s Design

In this section, we explain how Tsumiki’s design applies the principles in the previous section. Figure 1 overviewes the resulting testbed components and interfaces, represented by boxes and arrows respectively. In Section 4.1, we apply principles P1 and P2 to a set of desirable testbed capabilities. In Section 4.2, we discuss the impact of principles P3 and P4 on component interfaces. Specifically, Section 4.2.1 explains the inter-component interfaces (solid arrows in Figure 1), and Section 4.2.2 describes the human-facing interfaces between the components and human actors (dashed arrows in Figure 1). Finally, Section 4.3 explains how policies can change a component’s behavior without changing its realization.

4.1 Components and their purposes

We iteratively apply principles P1 and P2 to delineate seven Tsumiki components with specific networked testbed capabilities (Table 1). Starting with a monolithic testbed, each step builds on prior steps. To further explain these components, we provide examples from existing testbeds.

1. Supporting remote experimentation. In a distributed testbed an experimenter needs to manage experiments running on remote devices from a local machine. For example, in GENI 4 an experimenter can control her experiment remotely using Omni 25 or jFed 26. Both tools execute on an experimenter’s local computer, and provide access to hardware resources using her testbed credentials. Following P1, we introduce an experiment manager component in the Tsumiki model.

It is common for testbed software to be divided into device software and infrastructure services. For example, PlanetLab Central develops software that runs on devices maintained by institutions around the world. In this case, the device owners are distinct from the testbed provider. We further subdivide device software and infrastructure services in the steps below.

2. Device owner retains control over device use. When the testbed hardware resources are supplied by device owners, it is important to allow these owners to manage any software running on their devices. Managing, in this case, would include installing and removing the device software, starting and stopping the software, etc. In PlanetLab’s case, an institution’s IT staff controls the local PlanetLab nodes. These staff members add or remove nodes, and control the bandwidth that a node can consume. In Dasu 12, end-users can disable experiments using the Dasu BitTorrent extension. By principle P1, we identify the need for a device manager component as part of the device software. The device manager allows an owner to enable and disable the device software, controls the software installed on each device, and keeps it updated as new versions are released.

3. Sharing a device between multiple experiments. While testbeds like Emulab enable a researcher to obtain dedicated resources, many testbeds multiplex devices among multiple experimenters. This happens in situations where hardware resources are limited. For instance, on PlanetLab, multiple experimenters can run virtual machines on the same hardware. Other platforms that support such resource sharing include SPLAY, Dasu, and Fathom. Experimenters sharing a device may not trust one another and also may not be trusted by the testbed developers. Applying principle P1, we delineate a sandbox component that is part of the device software. The sandbox enforces a set of security restrictions to isolate experimenters’ code from one another and to prevent the code from harming the device.

4. Securing remote access. An experimenter accessing a sandbox on a device must have a trust relationship with that sandbox. In PlanetLab, a node manager mediates an experimenter’s access to sandboxes on a node through ssh-based authentication 27. In SPLAY, access to a device is managed by an administrator who configures a set of access keys 28. Applying principle P1, we delineate our final device software component called the resource manager. Principle P2 would also lead to this separation, because in many cases, an experimenter may want to run a sandbox locally, without network connectivity. The resource manager controls who can access the sandboxes on the device, and communicates with experiment manager (see step 1).

The software on a device is now composed of three components: a device manager, a resource manager, and one or more sandboxes (Figure 1). Next, we introduce the components that make up the testbed infrastructure.

5. Supporting testbed providers who are not testbed developers. In a common testbed instantiation, the testbed provider is also the testbed developer: in Dasu and Fathom
a device runs a fixed software stack created by the testbed provider. Sometimes, one group wishes to build a testbed using the software provided by another group, while retaining control over how the hardware resources are allocated. PlanetLab supports this model through a federation relationship between PlanetLab Europe and PlanetLab Japan. Different groups utilize the same PlanetLab software to manage resources in their region of PlanetLab. This also happens in Emulab where independent site providers predominantly run unmodified code maintained by the core Emulab team. Thus sites around the world need only provide and manage hardware, while benefiting from the Emulab team’s software development efforts. To support this scenario, principle P1 requires us to delineate an installer builder component. This component enables a testbed developer to provide software updates or customization to devices whose hardware is managed by a separate testbed provider.

6. Tracking devices. Both the testbed infrastructure and experimenters must have a way of locating remote devices, especially if they are mobile and may frequently change their IP addresses. Devices in Dasu use a configuration service and experiment administration service to announce their availability and characteristics like their geographic location. However, not every testbed needs such a tracking service. By principle P2 we introduce an optional lookup service to track device locations.

7. Allocating device resources. Experimenters typically gain access to devices through a trusted intermediary. In PlanetLab, PlanetLab Central mediates experimenter access to the available nodes, and creates slices on experimenters’ behalf. Similarly, SPLAY’s splayctl is a trusted entity that controls the deployment and execution of applications. Such trusted intermediaries are necessary to coordinate access among many experimenters in a large testbed. However, a testbed with a handful of experimenters may not need such a service. Following principle P2, we introduce the clearinghouse, an optional component in the testbed infrastructure. This component serves as the experimenter-facing mechanism for acquiring and managing device resources.

Monitoring. Additionally, each component includes additional monitoring and reporting logic. For example, the sandbox monitors resource consumption of experimenter’s code, the device manager monitors the overall resources on the device, the lookup service monitors device connectivity/availability, and the clearinghouse aggregates monitoring information from devices and monitors overall usage by experimenters.

Above, we used principles P1 and P2 to justify the set of Tsumiki components. An application of P1 and P2 on a different set of capabilities may derive different components. We will show in Sections 4.2.2, 4.2.3, and 4.2.4 that Tsumiki’s set of components is effective in practice, with several groups besides the authors of this paper using Tsumiki to build a variety of testbeds. We also consider other testbeds and how they compare/contrast with Tsumiki in Section 11. Specifically, Table 11 maps 20 testbeds from prior work into the Tsumiki model.

### 4.2 Tsumiki interfaces

In addition to Tsumiki components, its interfaces also play a crucial role. We now overview Tsumiki’s interfaces with a focus on how principles P3 and P4 influenced their design.

At first glance, principle P3 may seem incompatible with Tsumiki’s goal of supporting a variety of testbeds: how does one allow a diversity of realizations, when the interfaces are non-extensible? Our design reconciles this discrepancy through extensive use of opaque parameters (Section 4.2.1).

This same philosophy was used for the opaque rspec field in the GENI AM API. However, we apply it more extensively to the full range of interfaces, such as uploading and downloading files, running experiments, etc. Supporting a diversity of component realizations is also directly aided by P4, which mandates that human-visible interface must be flexible (Section 4.2.2.

Figure 1 illustrates how Tsumiki components communicate using these interfaces.

Several other efforts, notably GENI [4], have proposed common APIs for networked testbeds. However, this prior work targets federation and not making it easier to develop new testbeds. For example, projects like Emulab, ORBIT, and PlanetLab, that are part of GENI, consist almost entirely of disjointed code bases. We further detail prior work in this space in our related work (Section 11).

#### 4.2.1 Non-extensible inter-component interfaces (P3)

The use of opaque parameter values is key in designing interfaces that are non-extensible, yet able to support a variety of testbeds. This means many Tsumiki components receive or pass information through an interface that is opaque to them. That is, the component often does not interpret the data, or interprets just enough of it to perform a limited set of operations.

Another important concept in Tsumiki is the use of different keys. Every sandbox in Tsumiki has two kinds of keys, which are provided to the installer builder and controlled by the resource manager. One is a set of user keys and one is an admin key. A user key grants the key holder, typically an experimenter, access to run code in a sandbox. The admin key grants the key holder, usually the clearinghouse, the privilege to perform actions, such as changing the set of user keys, and allocating resources between the sandboxes on a device.

Below we outline the inter-component interfaces, and illustrate the usage of opaque parameter values, with resource allocation serving as a running example.

**Installer builder interface.** The device manager, resource manager, and sandbox are packaged together into an installer by the installer builder. This installer is customized to include the appropriate user and admin keys. New custom installer builder components can be set up if a group

---

2. As an example, here is an installer created by the installer builder in the Seattle testbed: [https://seattleclearinghouse.poly.edu/download/flibble/](https://seattleclearinghouse.poly.edu/download/flibble/)
The installer builder calls provide interfaces to allow, mostly the clearinghouse, to (1) package device software into an installer (with configuration information that will be discussed in a moment), and (2) provide this installer to a device owner who wishes to download and install it. The installer builder prepares an installer based on a device profile (e.g., an Android tablet) and configures the installer file (e.g., a .tar.gz) with public keys, and with a designated percentage of resources from the device that each key holder should be granted. The meaning of the keys and percentages is opaque to the installer builder: a party (often the clearinghouse) provides this as a text file that the installer builder merely adds at a pre-specified location on the device, which will later be used by the device manager. The installer builder calls are shown in Table 2.

TABLE 2: Installer builder calls.

<table>
<thead>
<tr>
<th>Installer builder call</th>
<th>Function</th>
<th>Interface to other component(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>build_installers (sandboxes, user_data)</td>
<td>Builds an installer for all platforms the installer builder supports, using the given sandboxes and user information.</td>
<td>Often used by clearinghouse’s request_install() call (described later) to build a new installer for a specific request.</td>
</tr>
<tr>
<td>get_urls(build_id)</td>
<td>Returns a set of URLs from which installers for the given build_id may be downloaded.</td>
<td>Often used by clearinghouse’s request_install() call to generate a URL for the newly built installer.</td>
</tr>
</tbody>
</table>

Remote resource manager interface. A device may be shared between experimenters, and also between several testbeds. However, the testbed providers may not share explicit trust between each other. To facilitate shared control of devices across experimenters and testbeds, Tsumiki uses the resource manager to mediate various types of access to the sandboxes on a device. The resource manager exposes three categories of calls to the clearinghouse and experiment manager: (1) calls available only to admins (typically the clearinghouse), (2) calls available to admins and users (clearinghouse and experimenters), and (3) public calls. These calls are grouped by the admin or user permission [29]. Table 3 overviews all of the calls, grouped by category.

(1) Calls available only to admins. The first category includes calls to change admin/user information, and calls to split/join sandboxes. The calls for changing admin/user information, which we call access control operations (the first three calls in Table 3), let the admin set the keys that are allowed to perform admin and user calls to the resource manager. The calls for splitting/joining sandboxes (the fourth and fifth calls in the table) are achieved through controlling the allocation of resources on the device. Both kinds of calls provide interfaces to the clearinghouse, as shown in the table, so that the clearinghouse can make calls to the resource manager that ultimately controls the allocation of resources on the device among the sandboxes.

(2) Calls available to admins and users. This class of resource manager calls manipulates the experimenter program and state in a sandbox. It provides interfaces to both the clearinghouse (a typical admin) and experiment manager (a typical user). For each sandbox on the device the resource manager maintains a state machine, as shown in Figure 2. It regularly checks the status of a sandbox and updates the corresponding state machine.

In addition to controlling the experimenter’s program, the resource manager can use calls to manipulate the file system in the sandbox and to retrieve the sandbox console log. These calls can be used by an experiment manager, e.g., when it calls upload filename, download filename, show files, show log, etc. The complete list of calls with arguments are listed in the middle section of Table 3.

(3) Public calls. These calls allow anyone to inspect the status of a sandbox and to determine the resources associated with it. These calls are listed at the bottom of Table 3. They are usually used by the clearinghouse or an experiment manager.

Device manager to resource manager interface. The device manager lets the device owner control the resource manager and the sandboxes. The device manager can stop or start these components, control the ports and resources (e.g., to limit the percent of the CPU that testbed code is allowed to use) used by these components, and retrieve updates from the installer builder. The device manager creates the initial resource allocation to partition the resources between the keys (i.e., the admin or users who hold these keys) using the percentages listed in the installer. The percentage values are converted into resource quotas, based on the capabilities of the hosting machine [30]. However, to the device manager, the keys themselves are opaque values. The device manager calls are shown in Table 4.

Sandbox to resource manager interface. This interface allows the resource manager to control sandboxes, directed by commands received from a remote experiment manager. The sandbox contains and executes experimenter code under the direction of the resource manager. The default

3. After an installation, the sandbox is initialized into the Fresh state. The StartSandbox() resource manager call changes its state to Started; when an experimenter’s program exits, the sandbox moves into Terminated state, and so on. If the sandbox has not recently updated its state (e.g., due to load from other processes on the system), then the resource manager considers the sandbox state to be Stale. Finally, the sandbox can be reset from any state back to Fresh through the ResetSandbox() resource manager call.
sandbox and resource manager interface has just four calls: the resource manager can start and stop the sandbox, retrieve the status of the sandbox by calling status (the resulting status can be Fresh, Started, Terminated, Stopped, or Stale as in Figure 2), and retrieve the log from the sandbox (getlog). The sandbox enforces the resource quotas it receives from the resource manager [50].

Note that the meaning of resource quotas is opaque to the resource manager. It simply performs addition and subtraction to divide the resource quotas between sandboxes, as instructed by the admin controlling those resources. The sandbox enforces the resource quota in experimenter code.

**Lookup service interface.** Like traditional DHT services, the lookup service has a simple get/put interface to retrieve values associated with keys, and to associate keys with values, respectively. This interface is used by the resource manager to advertise device availability, while the clearinghouse and the experiment manager use this interface to locate devices in the testbed. This is especially important for mobile devices, which frequently change their IP address.

### Table 3: Resource manager calls.

<table>
<thead>
<tr>
<th>Resource manager call</th>
<th>Function</th>
<th>Exemplary use by other component(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StartSandboxes(sandbox, platform, args)</code></td>
<td>Begins running a sandbox on a given programming platform (e.g. Repy, Table 5, Docker) with a set of arguments.</td>
<td>Experiment manager starts a Repy experiment by calling run program [args].</td>
</tr>
<tr>
<td><code>StopSandboxes(sandbox)</code></td>
<td>Stops the execution of a sandbox.</td>
<td>Experiment manager performs stop on sandbox.</td>
</tr>
<tr>
<td><code>ResetSandboxes(sandbox)</code></td>
<td>Stops running sandbox, removes sandbox file system, and resets log.</td>
<td>Experiment manager resets an experiment. Clearinghouse (in release resources(auth, list_of_handles)) resets sandbox when expired or released by experimenter.</td>
</tr>
<tr>
<td><code>AddFileToSandboxes(sandbox, filename, filedata)</code></td>
<td>Creates (or overwrites if it exists) a file called filename in the sandbox with contents filedata. Fails if file system is too small.</td>
<td>Experiment manager performs upload filename.</td>
</tr>
<tr>
<td><code>RetrieveFileFromSandboxes(sandbox, filename)</code></td>
<td>Downloads a file named filename from the sandbox to the experimenter's local machine.</td>
<td>Experiment manager performs download filename.</td>
</tr>
<tr>
<td><code>DeleteFileInSandboxes(sandbox, filename)</code></td>
<td>Deletes a file called filename from the sandbox.</td>
<td>Experiment manager performs delete filename.</td>
</tr>
<tr>
<td><code>ListFilesInSandboxes(sandbox)</code></td>
<td>Displays all the files in the sandbox.</td>
<td>Experiment manager performs show files.</td>
</tr>
<tr>
<td><code>ReadSandboxLog(sandbox)</code></td>
<td>Returns the sandbox's console log (stdout and stderr).</td>
<td>Experiment manager performs show log.</td>
</tr>
</tbody>
</table>

(3) **Public calls**

<table>
<thead>
<tr>
<th>Device manager call</th>
<th>Function</th>
<th>Exemplary use by other component(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetSandboxStatus()</code></td>
<td>Returns the sandbox name, admin key, status, user key(s), and admin information for all sandboxes on a device.</td>
<td>Experiment manager performs browse and list. Clearinghouse (in check online sandboxes daemon) queries the status of a sandbox.</td>
</tr>
<tr>
<td><code>GetSandboxResources(sandboxname)</code></td>
<td>Determines the resources associated with a sandbox.</td>
<td>Clearinghouse (in check new install daemon) registers new installs. Experiment manager performs show resources.</td>
</tr>
</tbody>
</table>

### Table 4: Device manager calls.

<table>
<thead>
<tr>
<th>Device manager call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StopResourceManager()</code></td>
<td>Stops the resource manager, particularly when an update is available.</td>
</tr>
<tr>
<td><code>StartResourceManager()</code></td>
<td>Used during installation to benchmark the system resources. This step is performed once, and generates a resource file to be used by resource manager's GetSandboxResources() call.</td>
</tr>
<tr>
<td><code>AutoUpdate()</code></td>
<td>Runs in the background, sleeps between 30 and 60 minutes, and checks with the installer builder for updates. If available, downloads the update for the installer, and replaces the older version.</td>
</tr>
</tbody>
</table>

### Table 5: Lookup service calls.

<table>
<thead>
<tr>
<th>Lookup service call</th>
<th>Interface to other components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>Used by a resource manager as put(admin_pubkey, ip:port), and put(user_pubkey, ip:port) for admins and users.</td>
</tr>
<tr>
<td><code>value=get(key)</code></td>
<td>Used by clearinghouse's check new install daemon() and check online sandboxes daemon(), which call ip:port=get(user_pubkey) to look for newly-installed and online sandboxes. This call is also used by the experiment manager's browse command to locate available devices associated with the experimenter's key.</td>
</tr>
</tbody>
</table>

The lookup service interface to the other components (e.g., resource manager, clearinghouse, and experiment manager) operates as in Table 5.

For increased fault tolerance and availability, Tsumiki supports seven distinct realizations of the lookup service. These range from distributed hash tables, such as OpenDHT [31], to centralized variants that support TCP and UDP protocols, NAT traversal, and other features. The installer builder bundles software that indicates the lookup services to be used by devices participating in the testbed.

Over our eight years of experience, we have made just one change to our inter-component interfaces. We changed the resource manager call that instantiates a sandbox to include an argument specifying the sandbox type. This change was necessary because we did not foresee that it may be desirable for a resource manager to allow an experimenter to start different types of sandboxes on the same device.

### 4.2.2 Flexible human-visible interfaces (P4)

Principles P3 and P4 allow Tsumiki to strike a balance between interoperability and the over-prescription of co-

---

TABLE 4: Device manager calls.

<table>
<thead>
<tr>
<th>Device manager call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StopResourceManager()</code></td>
<td>Stops the resource manager, particularly when an update is available.</td>
</tr>
<tr>
<td><code>StartResourceManager()</code></td>
<td>Used during installation to benchmark the system resources. This step is performed once, and generates a resource file to be used by resource manager’s GetSandboxResources() call.</td>
</tr>
<tr>
<td><code>AutoUpdate()</code></td>
<td>Runs in the background, sleeps between 30 and 60 minutes, and checks with the installer builder for updates. If available, downloads the update for the installer, and replaces the older version.</td>
</tr>
</tbody>
</table>

### Table 4: Device manager calls.

<table>
<thead>
<tr>
<th>Device manager call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>StopResourceManager()</code></td>
<td>Stops the resource manager, particularly when an update is available.</td>
</tr>
<tr>
<td><code>StartResourceManager()</code></td>
<td>Used during installation to benchmark the system resources. This step is performed once, and generates a resource file to be used by resource manager’s GetSandboxResources() call.</td>
</tr>
<tr>
<td><code>AutoUpdate()</code></td>
<td>Runs in the background, sleeps between 30 and 60 minutes, and checks with the installer builder for updates. If available, downloads the update for the installer, and replaces the older version.</td>
</tr>
</tbody>
</table>

### Table 5: Lookup service calls.

<table>
<thead>
<tr>
<th>Lookup service call</th>
<th>Interface to other components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>Used by a resource manager as put(admin_pubkey, ip:port), and put(user_pubkey, ip:port) for admins and users.</td>
</tr>
<tr>
<td><code>value=get(key)</code></td>
<td>Used by clearinghouse’s check new install daemon() and check online sandboxes daemon(), which call ip:port=get(user_pubkey) to look for newly-installed and online sandboxes. This call is also used by the experiment manager’s browse command to locate available devices associated with the experimenter’s key.</td>
</tr>
</tbody>
</table>

The lookup service interface to the other components (e.g., resource manager, clearinghouse, and experiment manager) operates as in Table 5.

For increased fault tolerance and availability, Tsumiki supports seven distinct realizations of the lookup service. These range from distributed hash tables, such as OpenDHT [31], to centralized variants that support TCP and UDP protocols, NAT traversal, and other features. The installer builder bundles software that indicates the lookup services to be used by devices participating in the testbed.

Over our eight years of experience, we have made just one change to our inter-component interfaces. We changed the resource manager call that instantiates a sandbox to include an argument specifying the sandbox type. This change was necessary because we did not foresee that it may be desirable for a resource manager to allow an experimenter to start different types of sandboxes on the same device.

### 4.2.2 Flexible human-visible interfaces (P4)

Principles P3 and P4 allow Tsumiki to strike a balance between interoperability and the over-prescription of com-

---

TABLE 5: Lookup service calls.

<table>
<thead>
<tr>
<th>Lookup service call</th>
<th>Interface to other components</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>put(key, value)</code></td>
<td>Used by a resource manager as put(admin_pubkey, ip:port), and put(user_pubkey, ip:port) for admins and users.</td>
</tr>
<tr>
<td><code>value=get(key)</code></td>
<td>Used by clearinghouse’s check new install daemon() and check online sandboxes daemon(), which call ip:port=get(user_pubkey) to look for newly-installed and online sandboxes. This call is also used by the experiment manager’s browse command to locate available devices associated with the experimenter’s key.</td>
</tr>
</tbody>
</table>
TABLE 6: External Repy sandbox interface.

<table>
<thead>
<tr>
<th>File system</th>
<th>Threading</th>
</tr>
</thead>
<tbody>
<tr>
<td>openfile</td>
<td>file.close</td>
</tr>
<tr>
<td>file.readat</td>
<td>file.writeat</td>
</tr>
<tr>
<td>listfiles</td>
<td>removefile</td>
</tr>
</tbody>
</table>

Network
- gethostbyname
- getmiopenconnection
- listenforconnection
- tcpserverSocket.getconnection
- sendmessage
- listenformessage
- udpserverSocket.getmessage
- socket.send
- socket.recv
- socket.close

TABLE 7: Commands for the default experiment manager in Tsumiki (seash) and the corresponding resource manager calls. The experiment manager commands are issued by an experimenter from his local host.

<table>
<thead>
<tr>
<th>Experiment manager command</th>
<th>Function</th>
<th>Invoked resource manager call(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>browse</td>
<td>Finds sandboxes available to</td>
<td>GetSandboxResources, GetSandboxStatus</td>
</tr>
<tr>
<td></td>
<td>the experimenter using the</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lookup service</td>
<td></td>
</tr>
<tr>
<td>list</td>
<td>Prints sandbox status</td>
<td>GetSandboxStatus</td>
</tr>
<tr>
<td></td>
<td>(Fresh, Started, Terminated,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stopped, Stale)</td>
<td></td>
</tr>
<tr>
<td>upload filename</td>
<td>Uploads file</td>
<td>AddFileToSandbox</td>
</tr>
<tr>
<td>download filename</td>
<td>Downloads file</td>
<td>RetrieveFileFromSandbox</td>
</tr>
<tr>
<td>show files</td>
<td>Lists all files</td>
<td>ListFilesInSandbox</td>
</tr>
<tr>
<td>show log</td>
<td>Prints console log</td>
<td>ReadSandboxLog</td>
</tr>
<tr>
<td>start program-name [args]</td>
<td>Starts running the program</td>
<td>StartSandbox</td>
</tr>
<tr>
<td></td>
<td>(with optional arguments)</td>
<td></td>
</tr>
<tr>
<td>stop</td>
<td>Stops the running program</td>
<td>StopSandbox</td>
</tr>
<tr>
<td>reset</td>
<td>Resets sandboxes</td>
<td>ResetSandbox</td>
</tr>
</tbody>
</table>

TABLE 8: Clearinghouse calls.

<table>
<thead>
<tr>
<th>Clearinghouse call</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire_resources</td>
<td>Given a resource specification, this call acquires resources for the experimenter, identified by an authentication structure auth. This call can also be used by an experiment manager to acquire sandboxes.</td>
</tr>
<tr>
<td>release_</td>
<td>Releases previously acquired by the experimenter. The list of handles indicates the sandbox(s) to be released. This call can also be used by an experiment manager to release sandboxes.</td>
</tr>
<tr>
<td>renew</td>
<td>Renew(s) resources previously acquired by the experimenter. The list of handles indicates the sandbox(s) to be renewed. By default, sandboxes are renewed for seven days.</td>
</tr>
<tr>
<td>get_resource_info</td>
<td>Returns a list of resources currently acquired by the experimenter.</td>
</tr>
<tr>
<td>request_installer</td>
<td>The API equivalent of a device owner clicking a link to request an installer for his/her device. When receiving this request, the clearinghouse requests the installer builder to include the clearinghouse’s keys within the installer, and returns the generated URL for download.</td>
</tr>
</tbody>
</table>

The most widely used tool is an interactive shell called seash. Using seash, an experimenter can communicate with remote devices, upload and run experiments, collect experiment results, and do all of these in parallel. Note that one can group a number of sandboxes together and issue commands to the group. A set of common experiment manager commands (implemented by seash) and the associated (remote) resource manager calls are listed in Table 7. As shown in the table, seash supports the commands that handle both local state and experiment setup, and the actual interactions with sandboxes.

Device manager interface. The device manager enables an owner to control the Tsumiki software running on their devices. Its external interface can vary significantly. For example, the Seattle testbed’s device manager allows a device owner to control the available network interfaces, whereas a device manager in Sensibility Testbed controls whether sensors, like GPS and accelerometer, are available to experimenters. Furthermore, these interfaces are implemented as a command line console and a mobile app, respectively.

Clearinghouse interface. Experimenters may want to share a pool of devices. A natural way to coordinate this sharing is through a clearinghouse that allocates resources according to a designated policy. While a clearinghouse is not mandatory, it helps a testbed to scale as the number of devices increases.

Typically, experimenters interact with the clearinghouse through a website, which allows experimenters to register for an account to gain access to a pool of device sandboxes. For example, the Social Compute Cloud [34] and the Tsumiki platform [35] customized the clearinghouse interface to show device resources in the experimenters’ social network. Sensibility Testbed uses a custom clearinghouse interface through which an experimenter submits her IRB policies for sensor component behavior. In this section we present a broad view of some of the implications of P4. We illustrate our point through examples of alternatives of all the human-visible interfaces in Tsumiki.

Sandbox interface (human-visible). The purpose of the sandbox is to contain and securely execute experiment code. The sandbox exposes an API to experiment programs. Through this interface, an experimenter can access resources on a remote device. There are a variety of sandbox realizations, including a Google Native Client-based sandbox, and a Restricted Python (Repy) sandbox that executes a subset of the Python language [32]. These realizations vary in such details as whether they execute binaries or code in a specific programming language. The Repy sandbox, widely used in Tsumiki realizations, provides a set of system calls available to experiment code, as listed in Table 6. A detailed description of these calls can be found online [33]. Another type of sandbox based on Docker is described in Section 6.

Experiment manager interface. An individual experimenter can use an experiment manager locally to control an experiment deployed on remote devices. However, testbed users often have different use cases, requiring different experiment manager realizations. Existing realizations include interfaces to accommodate automated deployment of long-running experiments, fast deployments through a parallel shell interface, and a GUI-based interface.

The most widely used tool is an interactive shell called seash. Using seash, an experimenter can communicate with...
data access. Using this interface, experimenters indicate sensor types, and required sensor access policies [16].

Clearinghouse human-visible interfaces can vary significantly across testbeds. For example, in the Seattle clearinghouse, every call is accessible through the backend of an authenticated web page (listed in Table 8). The interfaces in the top half of the table are called on behalf of an experimenter or a device owner, upon their request. Additionally, the clearinghouse runs two daemons periodically in the background, as shown in the bottom half of Table 8 to check for sandboxes that are online.

4.2.3 Overview through testbed scenarios

We introduced the inter-component interface (Section 4.2.1) and human-visible interface (Section 4.2.2). We now walk through two testbed scenarios to show how different components in Tsumiki interact, and how trust boundaries (from principle P1) are realized in practice.

We first describe the scenario in which Alice, a device owner, contributes her device to the testbed. Then, we describe a scenario in which Bob, an experimenter, runs his experiment on a Tsumiki-based testbed.

Scenario 1: Alice contributes a device to the testbed (Figure 3). Device installation. When Alice decides to contribute her device to a testbed, she downloads a testbed-specific installer through the clearinghouse website. The clearinghouse creates this installer through the installer builder with the build_installers(node join pubkey) call. The node join pubkey is unique to the testbed and is used by new devices to advertise themselves through the lookup service. The installer builder packages this key with the device manager, resource manager, and sandbox into a set of platform-specific installers. During installation, the device manager on Alice’s device benchmarks the device system, and saves the result to use for later configuration.

Device discovery. Since the installer included node join pubkey, the resource manager on Alice’s device has this key in a user sandbox after installation. When the resource manager starts, it advertises the IP:port of the sandboxes on Alice’s device, to the lookup service by calling put(node join pubkey, IP:port). The clearinghouse periodically calls get(node join pubkey) on the lookup service to discover new devices installed with the testbed-specific installer. After discovering Alice’s device, the clearinghouse contacts the resource manager on her device, and instructs the resource manager to configure the sandboxes according to the benchmark results.

Scenario 2: Bob runs an experiment (Figure 4). This scenario includes several steps. First, Bob needs to request resources from the device(s) in a testbed, before he can run code in a device sandbox. When the experiment is finished, Bob can then release the acquired resources.

Experimenter requests resources. When Bob registers for an account with the clearinghouse, he provides a public key, bob pubkey, such that Bob has the corresponding private key bob privkey. After finding a suitable sandbox upon Bob’s request, e.g., a sandbox on Alice’s device (through the device discovery step in Scenario 1), the clearinghouse contacts the resource manager on her device and instructs the resource manager to change the user of this sandbox to Bob, by calling ChangeUsers(sandbox, bob pubkey). The resource manager then issues a put(bob pubkey, IP:port) call in the lookup service, which allows Bob to look up the sandbox he requested. This scenario is shown in Figure 4(a).

Experimenter runs code. Bob then uses the experiment manager to control the sandbox on Alice’s device from his local machine. Bob loads his public and private keys into the device, and then requests resources from the device(s) in the testbed. When the experiment is finished, Bob can then release the acquired resources.

Experimenter releases the acquired resources. When Bob is finished with his experiment, he may release his sandbox explicitly through the clearinghouse. To prevent experimenters from holding onto unused resources, the clearinghouse may also implement a policy to expire acquired sandboxes after a timeout (i.e., resources must be periodically renewed to prevent expiration). In both cases the clearinghouse issues a ChangeUsers(sandbox, bob pubkey) call on the resource manager on Alice’s device to remove bob pubkey from the sandbox, and resets the sandbox with ResetSandbox().
4.3 Customizing testbeds with policies

Tsumiki supports further customization of testbeds, such that even two testbeds that use the same component realizations do not need to behave identically. This is achieved through Tsumiki policies that change the behavior of a component without requiring a new component realization. Tsumiki enforces policies in three components: the clearinghouse, resource manager, and sandbox.

4.3.1 Clearinghouse Policies

A sandbox admin, typically the clearinghouse, can affect the resource allocation among experimenters. The sandbox admin can also use the appropriate API and restrict available resources for the sandbox. We briefly describe four clearinghouse policies that we have implemented:

- **Incentivizing participation.** Testbed providers can implement clearinghouse policies to incentivize device owners to participate in the testbed. For example, Seattle (Section 4.2.3) uses a policy where device owners donate resources on their devices in exchange for resources on other devices. For each device that runs Seattle, the owner gets access to sandboxes on 10 other devices on the Seattle testbed.

- **Restricting the experimenters who obtain access to the clearinghouse.** Sensibility Testbed (Section 5) restricts logins to experimenters by requiring IRB approval for their experiment. Social Compute Cloud (Section 7), which integrates with social network platforms such as Facebook, only allows logins from users with Facebook accounts, and includes policies to match experimenters with resources provided by people in their social network. By contrast, the Seattle testbed clearinghouse allows anyone to register for an account and login.

- **Resource expiration.** Experimenters may want to hold onto resources that they no longer use. A simple policy to discourage this is to have resources expire after a certain time period (i.e., the experimenter must re-acquire or renew the resources every so often).

- **Resource allocation between sandboxes.** The default policy in the clearinghouse of Tsumiki is to partition device resources equally between all sandboxes on a device. However, this policy could be changed. For example, a user can choose to allocate fewer resources to sandboxes that host long-running experiments, or grant more resources to sandboxes controlled by a specific set of experimenters.

4.3.2 Resource Manager Policies

When using the default Tsumiki sandbox, at install time device owners can set policies with the device manager and choose the amount of resources (CPU, memory, network bandwidth, etc.) they would allow testbed experimenters to access. The device manager then instructs the resource manager to apply these policies to control the sandboxes. We have implemented three kinds of resource manager policies, and also support others:

- **Fraction of device resources dedicated to the testbed.** The default device manager in Tsumiki benchmarks the device resources during installation and uses the results to configure the fraction of resources available to all sandboxes running on the device. The default policy is to use 10% of device resources. Another policy is to query the device owner for the fraction of their device resources that they want to dedicate to the testbed. This information is provided to the resource manager, which applies these policies to control the sandboxes. Sandboxes that try to exceed the set amount would be restricted.

4.3.3 Sandbox Policies

A sandbox applies the policies set by the device owner, clearinghouse, or resource manager. As an example, a device owner may wish to have experiment traffic routed over the WiFi (wlan0) interface. The resource manager may specify the amount of RAM that an experimenter’s sandbox may use, and a clearinghouse may wish to restrict UDP and TCP communications to prevent access to low numbered ports on remote devices. The device owner policies always override those of the clearinghouse and the resource manager. We have implemented two kinds of sandbox policies:

- **Containing sandbox traffic.** Policies of this kind limit the address and port ranges on which sandboxes can send and receive network traffic. For example, this can be used to whitelist some Tsumiki hosts that actively want to participate in an experiment, and to blacklist (and thus exempt) other Tsumiki hosts. In a practical deployment, this could mean that a device owner blacklists their LAN, while an experimenter can partition their set of sandboxes into subsets with limited inter-connectivity.

- **Preserving end-user privacy through blurring.** Sandbox implementations may expose potentially privacy-intrusive functions, such as reading out smartphone sensors that disclose the current physical location of the device (and thus its owner). A privacy policy changes the behavior of a call such that it can both reduce the precision of the return value, and the frequency at which new values are returned (thus blurring the data). For instance, the geolocation sensor could be blurred to return one value per hour, at an accuracy level corresponding only to the city the device is in, as opposed to a particular location within that city. To implement this, sandbox calls are transparently wrapped so that their capabilities are restricted, while the call signatures and semantics are unchanged.
5 Sensibility: A Mobile Devices Testbed

From this section on, we consider how Tsumiki’s design meets the goals in Section 2. To evaluate the utility and diversity of testbeds one can build, we first introduce Sensibility Testbed, a testbed constructed from Tsumiki components that runs on Android-based end-user devices.

5.1 Building Sensibility with Tsumiki

The Sensibility Testbed runs on Android devices and uses custom Tsumiki device manager and clearinghouse components. Sensibility extends the programming interface of the default Tsumiki sandbox, which includes access to only the basic file system, network, threading, etc. (Table 6). The new sandbox allows experimenter’s code to access sensor information without risking the security of end-user devices.

Reused and customized components. The Sensibility Testbed device software is customized and packaged as an Android application package (APK) on the Google Play store. This APK contains a native Android portion, a customized device manager, and an extended Tsumiki sandbox. When the app is started by a device owner, the native code initializes the sandbox, and starts the communication with the Sensibility Testbed clearinghouse. The native code also implements methods to access sensors and pass the sensor information to a sandboxed experiment program.

Reused and customized interfaces. The Tsumiki inter-component interfaces (Section 4.2.1) are not changed (from principle P3 in Section 3). The human-visible interfaces are customized as follows. First, Tsumiki’s Python-based sandbox programming interface is augmented with a sensor API. Experimenters can use calls like get_location, and get_accelerometer and dozens of others in their code. Second, as some experiments involve human subjects, the device manager interface is modified to request informed consent from device owners before the app can be installed, and to allow owners to opt out of any experiment. Finally, the clearinghouse is customized to enforce privacy policies when an experiment needs to access sensitive data. Therefore, an experimenter can conduct experiments without compromising ethical standards set by her institution.

Privacy policies. The customized clearinghouse plays an intermediate role between experimenters and device owners. When an experimenter registers at the clearinghouse, she needs to provide the IRB policies set by her institution. The clearinghouse translates and codifies each policy, often using pre-programmed policy code. It then instructs the sandboxes on remote devices to enforce these policies on experiment code. For example, to protect device owner’s location privacy, a policy can blur the location coordinates so that the experimenter’s code can observe only the device’s city. Details of the implementation can be found in [16, 39].

Effort in constructing Sensibility. Constructing Sensibility Testbed involved three major tasks: building a custom device manager, adapting the default Tsumiki sandbox to provide sensor access, and constructing a clearinghouse. We estimate that it took us a week of effort to build the device manager software. Most of this time was spent on refining the user interface. It took us several weeks to add the sensor calls to the sandbox and we spent approximately a month implementing a clearinghouse that obtains privacy policies from experimenters. The most time consuming process was waiting for IRB approval (#15-10751) from NYU.

5.2 Using Sensibility to replicate a study

To investigate whether testbeds constructed with Tsumiki can be used for non-trivial experiments, we used Sensibility Testbed and evaluate a previously published sensor-augmented WiFi protocol [17]. It is known that a WiFi device is much more likely to experience packet loss when the device is in motion. Therefore, the authors of [17] propose to use device sensors to detect if a device is moving and send explicit mobility notifications to the WiFi sender, so the sender can reduce data loss while the device is in motion.

The first algorithm, RapidSample, is designed for mobile wireless environments; the second, SampleRate [40], is designed for static wireless environments. Finally, the hint-aware adaptation algorithm switches between RapidSample and SampleRate when given a mobility hint from a smartphone accelerometer.

We used an Android Nexus 4 device as the receiver and a laptop running OS X 10.9.4 as the sender. The laptop sent a stream of 1,000-byte back-to-back UDP packets. The Android device acknowledged each packet and piggy-backed movement detection results on acknowledgment UDP packets. Both the laptop and the phone were connected to the same campus WiFi access point. In [17], the smartphone receiver was put in tethering mode to communicate with the laptop. In our experiment, the laptop and smartphone communicate via an intermediate WiFi router. The authors of [17] used the Click module for Linux, whereas we implemented the algorithms in the Sensibility Testbed’s sandbox. Due to these factors, the overall throughput achieved in [17] was different from our results.

Movement detection. The idea in [17] depends on accurate movement detection. Figure 5 shows the movement from experimenters. The most time consuming process was waiting for IRB approval (#15-10751) from NYU.

4. RapidSample algorithm (mobile). If a packet fails to get an acknowledgment, RapidSample switches to a lower sending rate and records the time of the failure. After achieving success at the target rate for a short period of time (30 ms in our implementation), the sender samples a higher rate. If the higher rate has not failed in the recent past (50 ms in our case), the sender switches to that rate. SampleRate algorithm (stationary). SampleRate changes its sending rate when it experiences four successive failures. The rate it switches to is selected from a set of rates that are periodically sampled. Over the course of 10 seconds, if the sampled rate exhibits better performance (lower packet loss and lower transmission time) than the current rate, SampleRate switches to this sampled rate.

Hint-aware adaptation algorithm (hybrid). This algorithm uses RapidSample for data transmission when the receiver is mobile, but SampleRate when the receiver is stationary. It relies on movement hints generated by the movement detection algorithm that are piggy-backed on acknowledgment packets to the sender, that are interpreted at the sender as a signal to switch between mobile and stationary algorithms. At the receiver, we use a smartphone accelerometer to detect movement.
detection results for our implementation; it corresponds to Figures 4 and 5 in [17]. Figure 5(a) shows the raw data from the three accelerometer axes. The data in Figure 5(b) is generated by calculating the acceleration magnitude along the three axes (including gravity). Figure 5(b) also shows the detection thresholds as dotted lines when the device is considered stationary (i.e., acceleration is between an upper and lower threshold). An acceleration or deceleration value outside these thresholds is considered a movement. Using the Sensibility Testbed’s sandbox, an undergraduate biology student implemented this algorithm in five hours.

**Protocol performance.** We ran three experiments to observe the performance of the three algorithms. In the static environment, the laptop and smartphone were stationary on an office desk. In the mobile environment, an experimenter walked in a hallway at a constant speed in direct line of sight to the WiFi access point. In the mixed static-mobile environment, the experimenter walked along the same hallway and also stopped at predefined locations for predefined time intervals. Our experiment results are shown in Figure 6, which corresponds to Figure 13 in [17].

Figure 5(a) shows that when devices are static, SampleRate performs better than RapidSample. In a mobile environment, as shown in Figure 5(b), both SampleRate and RapidSample experience throughput degradation because of packet loss. However, RapidSample outperforms SampleRate due to its fast response to high packet losses. In Figure 5(c), the hint-aware protocol out-performs both RapidSample and SampleRate. These results are consistent with results in [17].

A mistake found through replication. The replication experience with the Sensibility Testbed also led us to identify a mistake in the pseudo-code in Figure 7 in [17]. In “if (sample) then ... else”, the “sample ← 0” statement should appear in the if branch rather than in the else branch. This problem was confirmed by the authors (their Click implementation does not contain this bug). We found this problem experimentally when we noticed that RapidSample was unresponsive to packet loss.

### 6 DUCKY: DOCKER-BASED TESTBED

Tsumiki’s component-based design is intended to reduce systems researchers’ efforts in developing new testbeds, and to improve the compatibility of testbed components. In this section we describe Ducky, an early-stage prototype testbed that uses the Docker container [41] as the sandbox, and other default Tsumiki components. Docker is a tool that packages an application and its dependencies into a container that can run on any Linux host. Docker is supported by major commercial cloud providers. We expect that a testbed based on Docker containers would be of interest to systems researchers, and to our knowledge no such testbed exists.

Ducky uses Docker as the sandbox component and makes minimal changes to the other default Tsumiki components. As a proof-of-concept, Ducky provides evidence that Tsumiki’s design enables researchers to build useful testbeds that operate correctly with little effort. It took the first author of this paper, who had never used Docker, three hours of work to create Ducky. Furthermore, Ducky demonstrates that externally developed components can be compatible with default Tsumiki components using Tsumiki’s inter-component interfaces.

#### 6.1 Building Ducky with Tsumiki

Prior to the integration, we prepared a document detailing Docker 1.0.1 commands and the most likely corresponding Tsumiki calls. The first author then spent an uninterrupted time period on the integration task. Table 9 overviews the lines of code changed in each Tsumiki component during this integration and the rationale for the change.

<table>
<thead>
<tr>
<th>Component</th>
<th>LOC changed</th>
<th>Reason for change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sandbox</td>
<td>48</td>
<td>Integrate Docker containers with resource manager.</td>
</tr>
<tr>
<td>Resource manager</td>
<td>2</td>
<td>Add Docker as a sandbox type; add command line arguments.</td>
</tr>
<tr>
<td>Device manager</td>
<td>5</td>
<td>Install Docker package.</td>
</tr>
<tr>
<td>Experiment manager</td>
<td>10</td>
<td>Make Docker a default sandbox type.</td>
</tr>
<tr>
<td>Total</td>
<td>66</td>
<td>Support commands in containers.</td>
</tr>
</tbody>
</table>

Fig. 6: Protocol throughput in three different scenarios.

TABLE 9: Lines of code changed in Tsumiki to build Ducky.
Ducky is a basic proof of concept, which demonstrates that a Docker-based testbed can be built using Tsumiki components in a few hours. This fulfills our goals to reduce the development time and effort in building testbeds, and to support diverse testbeds. We estimate that it would take the first author an additional two days of effort to make the Ducky prototype complete, including advanced functionality like updating the Docker software automatically.

### 6.2 Performance evaluation

A basic measure of testbed performance is the time that it takes an experimenter to run a program, and to transfer a file to/from the device. We evaluate these two tasks in Ducky. We report results for a Ubuntu 14.04 machine on an Intel 2.4 GHz CPU, with 1GB of RAM. Both the experimenter host and remote device are located on a machine at NYU.

Table 10 summarizes the results.

**Table 10: Ducky performance over 30 runs.**

<table>
<thead>
<tr>
<th>Time to execute a process</th>
<th>Upload/download a file</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-time latency</td>
<td>Base latency</td>
</tr>
<tr>
<td>Mean (sec)</td>
<td>Mean (sec)</td>
</tr>
<tr>
<td>Median (sec)</td>
<td>Median (sec)</td>
</tr>
<tr>
<td>Standard Deviation (sec)</td>
<td>Standard Deviation (sec)</td>
</tr>
<tr>
<td>Upload speed</td>
<td>Download speed</td>
</tr>
<tr>
<td>Median (Mbps)</td>
<td>Median (Mbps)</td>
</tr>
<tr>
<td>Stanadard Deviation (Mbps)</td>
<td>Standard Deviation (Mbps)</td>
</tr>
<tr>
<td>5.4 sec</td>
<td>1.5 sec</td>
</tr>
<tr>
<td>3.4 ms</td>
<td>3.4 ms</td>
</tr>
<tr>
<td>31.1 Mbps</td>
<td>70.8 Mbps</td>
</tr>
</tbody>
</table>

**Executing a process.** We benchmarked the time to execute a single process in the Ducky sandbox. The one-time latency (including NTP setup and loading libraries) is about 5.4 s. The base latency is the time to execute the process (the ls command), including the handshake between the Ducky sandbox and the resource manager, and takes around 1.5 s. The time to start the sandbox is just 3.4 ms.

**File transfer.** We measured the time to upload and download a 10 MB file to/from the Ducky sandbox, resulting in bandwidths of 31.1 and 70.8 Mbps, respectively. Overall, the measured performance is equivalent to that of Seattle, a Tsumiki-based testbed that has been deployed for eight years (Section 5.1). Ducky thus not only operates correctly, but also has reasonable performance.

### 7 Social Compute Cloud

Thus far we have described testbeds based on Tsumiki that we have developed. In this section we illustrate that Tsumiki can be used by other researchers to develop real and useful testbeds. We describe how researchers from the University of Chicago and Karlsruhe Institute of Technology used Tsumiki to develop the Social Compute Cloud testbed [34, 35, 36].

**Testbed context.** Trust between people in a social network inspired several researchers to develop the Social Compute Cloud [34, 35, 36], which uses social networks to locate computational and storage resources in one's social circle. Figure 7 illustrates a use case where Social Compute Cloud provides a friend-to-friend storage service. In this system, the burden of hosting data is transferred to peer devices within one's social network. We detail how Tsumiki components were used to build Social Compute Cloud, and then use this testbed to measure the availability of resources within an individual’s social network.

**Reused or customized Tsumiki components.** The custom clearinghouse integrates with social networks and provides socially-aware resource allocations. This includes Facebook-based authentication, association between social and Tsumiki identities, and the use of Facebook access tokens to access a user’s social network and their sharing preferences. The researchers also developed new human-visible interfaces for users to define relationship-based sharing preferences. The custom clearinghouse uses these preferences and an external matching service to determine resource consumer–provider mappings. Following resource allocation, standard Tsumiki mechanisms are used.

**Experiences with integration.** The modifications took about one month. The development effort focused on implementing the social clearinghouse: accessing a user’s social graph and sharing preferences using the Facebook Graph API, a custom preference API, and caching of social graphs and preferences to optimize performance. The researchers also altered Tsumiki’s discovery process to retrieve socially connected pairs’ resources, rather than using location-based or random discovery methods. Finally, they extended the clearinghouse allocation process to make remote service calls to their existing preference matching service.

**Resources in a social network.** We evaluate Social Compute Cloud via the deployment of a friend-to-friend backup service. This service provides storage and computation resources from one’s social network. We describe the results from a five-day experiment deployed on Social Compute Cloud.

17 participants installed the device software of Social Compute Cloud on their personal devices: laptops, desktops, and smartphones. We then deployed an experiment in which each device sent a ping every 5 minutes to a server located at the researcher’s university. From this, we determined continuous periods of availability for all devices: 3 missed pings, or 15 minutes of inactivity, denoted the end of an availability period. The goal of the experiment was to evaluate the availability of devices owned by people in one of the authors’ university-based social circles on Facebook.

Figure 8(a) shows availability of data across all devices as a CDF, excluding five desktops with uninterrupted availability. The median continuous availability period across the devices was 2.5 hours. A friend-to-friend backup service can use this information for selective file placement. Figure 8(b) considers the device availability by hour, averaged over all five days (using medians). For each hour between 10 AM and 7 PM a median number of 10 devices was available. This can be used for scheduling data replication between...
The experiment showed the measured UDP inter-node latency. The user then installed these components to construct a testbed across three systems: a Linux desktop, a Mac laptop, and an Android tablet. Following this, (s)he deployed a provided experiment on this testbed. The experiment showed the measured UDP inter-node latency, the user-answered questions about the highest/lowest latency, and any connectivity issues between the devices.

Task 2. The user extended the Repy sandbox’s network API, which only supports TCP and UDP, to expose the native ping command to sandboxed programs. We use this task as a proxy for the kind of customization that a researcher would perform to create a testbed to fit their research purposes. After modifying the sandbox, users created a custom installer containing this sandbox. This step makes it possible to distribute and install the modified sandbox on other systems. For this the user had to configure a web-server to host the new installer. The user then downloaded and installed the new testbed software on a Linux desktop and a Mac laptop, and used it to test the native ping command against various machines on the Internet.

User study results. All seven users completed task 1 and six of them were able to complete task 2. The average completion time for task 1 was 30 minutes, and the maximum time spent was 39 minutes. The average time to complete task 2 was 36 minutes, with a maximum time of 53 minutes. This suggests that it is easy to use Tsumiki to construct a testbed, even by users who have no prior exposure to networked testbeds. The user who failed to complete task 2 missed a step in our instructions.

Despite our small sample, we were able to derive useful observations from the survey responses and interviews. Feedback during the semi-structured interview revealed that users would find the testbed useful in their own studies. Below we provide several representative quotes from the study participants.

Many of the users in the study commented on the flexibility provided by Tsumiki’s extensibility:

It [Tsumiki] is component based and by including your component you add it to the general framework . . . it’s lego-based. [U5]

I liked being able to specify which tools [like ping] can be invoked in a sandbox very easily. [U4]

Users also noted features that they would find useful in their research that Tsumiki currently lacks. We believe that the existing components in Tsumiki can be extended to handle such feature requests.

Being able to simulate routing infrastructure and routing delay would be something that I would look for . . . especially programmatically configuring delays between nodes. [U3]

When asked if they would use Tsumiki in their research, the majority of users agreed that it would be a good choice, particularly since they were now more familiar with it.

I feel comfortable with using Tsumiki right now, if I need [a testbed] then I will try Tsumiki first. [U5]

The ability to spin up a private network in a distributed fashion seems very handy. [U3]

I would probably consider Tsumiki [in my research] at least as an initial choice [of a testbed]. [U3]

The results of this user study show that Tsumiki fulfills our goal to reduce testbed development effort.

9 Other testbeds built with Tsumiki

Tsumiki has been used to build testbeds besides the ones examined in this paper. We now overview two such testbeds: Seattle, and Seattle On Access Router (SOAR).
components with minor changes. The most notable change was to reduce the disk space needed by removing extraneous files (such as support for platforms other than OpenWrt, the router operating system that SOAR runs on) to fit within the 16 MB flash storage on the platform. The component APIs did not change in any way. We also extended the sandbox with additional measurement functionality that is customized to the OpenWrt devices, such as the ability to monitor raw traffic passing through the router or to gather packet characteristics on the wireless interface, including delay, throughput, and jitter.

**Experiences with integration.** The needed changes were mostly performed over a two day period. Most changes involved two main areas. First, it was necessary to patch several portability issues with Tsumiki components. These were due to quirks in the OpenWrt environment. Second, it was necessary to shrink disk usage, which required building an installer to reduce the on-device footprint. It also required setting up a separate software updater to update the space-optimized version of software components run on the device. All other components operate and interoperate smoothly as the inter-component interfaces were unchanged.

By using Tsumiki, we can use any of the experiment managers to quickly and easily deploy our code. If one of the SOAR routers is moved to a different network, the lookup service will automatically be notified. Most notably, by running experiments in a Tsumiki sandbox, SOAR experiments are security and performance isolated. The use of Tsumiki components has made it easier to support SOAR's goals of non-interference with a user's network.

## 10 Limitations

As with any system, Tsumiki has certain limitations. We discuss the most important ones in this section.

**Our component set may not be canonical.** The seven Tsumiki components and their interfaces discussed here are based on our analysis of prior testbeds (Table 11), our use of the four principles outlined in Section 3, and our experiences with building testbeds. When considering a different set of testbed capabilities, applying these principles may lead to a different set of components. However, we believe that a future testbed with radically different assumptions could reapply our four principles to obtain a relevant set of components and interfaces.

**The Tsumiki set of actors/trust domains may be limited.** Similarly, our design assumed four actors: device owners, experimenters, testbed providers, and testbed developers. It is possible that future testbeds may require new actors. By principle P1, this may lead to different trust domains and thus different component derivation.

**Some features of Tsumiki make it a poor choice for certain environments.** Tsumiki's division into components provides a convenient way to construct diverse testbeds, but at a cost. This design has a performance penalty and may not be suitable for testbeds that must maximize performance. Therefore, having separable components implies that Tsumiki will perform worse than a monolithic testbed design. In an environment where high performance is critical, Tsumiki's design may have unacceptable overheads.
Artifact of component separation. Some Tsumiki components belong to different trust domains, which may lead to misconfigurations. For example, the clearinghouse is unaware of the device types, e.g., whether a device is mobile or not. It can assign this device, by accident, to an experimenter who should not access private information about the device owner. Meanwhile, this component-based design can also negatively impact Tsumiki’s efficiency and effectiveness on different platforms. For example, the device manager and resource manager are in different trust domains, and so it is convenient to implement these components as different OS processes. Similarly, each sandbox typically runs in its own process to provide performance isolation. This poses challenges for deploying Tsumiki on systems like iOS, in which an app runs in a single process.

Focus on userspace operation. Tsumiki strives to support the broadest range of devices possible, and does so by operating in userspace. Experiments that require low-level access, such as those that need a modified kernel, were not considered when designing Tsumiki’s interfaces. Thus we do not believe our current set of components and interfaces would be effective in this situation.

Tradeoff between performance isolation and performance. The experiment manager assumes a certain level of performance isolation provided by the sandbox and uses this to optimize experiment scheduling. For example, the experiment manager would schedule two CPU-heavy experiments on the same machine, expecting the sandbox to properly isolate them. This assumption does not always hold, e.g., when the sandbox is replaced by a version that does not support performance isolation. This leads to suboptimal performance when an experimenter expects the sandbox to isolate experiments properly. However, this assumption does not result in an incorrect operation.

11 Related work

Our design of Tsumiki components is drawn from experiences with existing testbeds and their approaches to reusing and extending functionality. Therefore, many of the concepts in Tsumiki’s design are due to prior testbed construction efforts by other groups. We conducted an analysis of existing testbeds and mapped them into the Tsumiki model (see Table 11). Note that this table is our interpretation of existing testbed construction, based on an analysis of the cited papers. Therefore, there may be inaccuracies as we are not intimately familiar with all design aspects of each system. Table 11 illustrates that most of the existing testbeds can be (at least partially) mapped into the Tsumiki model. This indicates that the Tsumiki component-based model is flexible enough to capture existing testbed designs.

In the rest of this section we consider prior approaches to extensibility and reuse, a key concern in Tsumiki.

Testbed functionality reuse. Many research groups have extended the functionality of existing testbeds. For example, VINI [67] extended PlanetLab with network programmability, and its design and implementation provided important lessons for GENI [4]. Flexlab [59] combines the characteristics of both Emulab and PlanetLab. SatelliteLab [11] includes nodes from PlanetLab as the core, and nodes on the Internet edge connected to the core. Similarly, Emulab [3] supports acquisition of resources on PlanetLab through fast slice instantiation. DETER [71] reuses much of Emulab’s code base, but provides specialized functionality for security experiments. Tsumiki’s design generalizes the concept of testbed reuse and builds on this rich prior work (Table 11).

Software similar to individual Tsumiki components. Many other testbed builders have built components that are similar to those of Tsumiki. For example, Xen [72] and the Lua-based sandbox in SPLAY [28] are examples of similar sandboxing techniques. Similarly, several tools exist for deploying and controlling remote software [23], [74]. Tsumiki decomposes all functionality in a testbed and provides well-defined interfaces so that individual components can be easily swapped out.

Federated testbeds. Today’s most prevalent model for testbed reuse is federation: testbeds interoperate to allow an experimenter to use a common authorization framework across several testbeds. These testbeds may also support a common resource allocation mechanism and common experimenter tools. For example, PlanetLab (PL) Europe and PL Japan pool resources within PL Central. Although the code base is shared between these testbeds, individual testbeds maintain their own nodes.

GENI [4] consists of testbeds that federate by supporting a common API, resource specification, and identification mechanisms. This allows a researcher to use a shared set of credentials and a common tool to access diverse testbeds with different implementations. However, federation does not make it easier to build new testbeds. Testbeds like Emulab, ORBIT, and PlanetLab, that are part of GENI, consist almost entirely of disjointed code bases. GENI focuses on interoperability of testbeds with different code bases, while Tsumiki’s focus is on enabling testbed builders to easily build new testbeds through code reuse. Our goal is not to allow existing testbeds to interoperate, but to provide a set of building blocks for constructing new testbeds.

Other component-based systems. Component-based systems have also been developed in other domains. The modular components in Flux OSKit [75] can be used to construct core OS functionality. OpenCom [76] provides a uniform programming model and a set of component libraries for defining software architectures. NETKIT [77] is a software component model for programmable networking systems. By contrast, Tsumiki provides network testbed components for building custom testbeds.

12 Conclusion

Building and deploying a new testbed is labor-intensive and time-consuming. New testbed prototypes will continue to be developed as new technologies appear and limitations of existing testbeds become apparent. This paper describes a set of principles that we have found effective in decomposing a testbed into a set of components and interfaces. The resulting design, Tsumiki, forms a set of ready-to-use components and interfaces that can be reused across platforms and environments to rapidly prototype diverse networked testbeds.

As our user studies have demonstrated, Tsumiki is easy to learn and use. Researchers have used Tsumiki to create a diverse set of testbeds and our user study with graduate
### TABLE 11: Existing testbeds mapped into the Tsumiki model. The testbeds are ordered alphabetically. There may be inaccuracies in this table as we performed the analysis based on the cited papers, and are not intimately familiar with the design aspects of each system. For entries reading “N/A”, we could not identify a component with that functionality in the testbed.

<table>
<thead>
<tr>
<th>Testbed</th>
<th>Sandbox</th>
<th>Resource Manager</th>
<th>Device Manager</th>
<th>Expansive manager</th>
<th>Installer builder</th>
<th>Caching house</th>
<th>Lookup service</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>BISmark</td>
<td>BISmark routers</td>
<td>sub server on BISmark router</td>
<td>Custom package management utilities on top of OpenWrt's built-in api</td>
<td>Data collection servers, BISmark-data-transmit</td>
<td>Package repository servers</td>
<td>User registration system (in firmware)</td>
<td>N/A</td>
<td>[7] [8]</td>
</tr>
<tr>
<td>BOINC</td>
<td>BOINC client</td>
<td>Python scripts, C++ interfaces, general preferences</td>
<td>BOINC installer tool</td>
<td>Scheduling servers, Data servers, Client GUI</td>
<td>Anonymous platform mechanisms</td>
<td>Project master URL</td>
<td>N/A</td>
<td>[27]</td>
</tr>
<tr>
<td>Chameleon Cloud</td>
<td>Physical or virtual machines and overlay network</td>
<td>Nova and Blazar</td>
<td>Interaction Manager</td>
<td>Ironic and Nova</td>
<td>Ironic</td>
<td>Blazar</td>
<td>Reference API</td>
<td>[29]</td>
</tr>
<tr>
<td>CloudLab</td>
<td>Slice and physical machines</td>
<td>or and global resource interfaces</td>
<td>CloudLab web interface</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>[21] [22]</td>
</tr>
<tr>
<td>CONPOSE</td>
<td>VM, skiv</td>
<td>Control software</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>[24] [25]</td>
</tr>
<tr>
<td>Dava</td>
<td>Dava client's experiment sandbox</td>
<td>Configuration service</td>
<td>Bismark-data-transmit</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>[21] [22]</td>
</tr>
<tr>
<td>DETER</td>
<td>Computing elements, Container</td>
<td>Virtualization engine</td>
<td>SEER (workbench), Experiment lifecycle manager (ELM)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>Discovery</td>
</tr>
<tr>
<td>DIMES</td>
<td>DIMES agent</td>
<td>Auto-update mechanism</td>
<td>Experiment planning system</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>[27]</td>
</tr>
<tr>
<td>Emulab</td>
<td>Virtual nodes, OpenVZ Linux containers, Xen, and physical machines</td>
<td>Global resource allocation</td>
<td>Emulab web interface, m</td>
<td>master host</td>
<td>Authorization structure</td>
<td>N/A</td>
<td>N/A</td>
<td>[3]</td>
</tr>
<tr>
<td>Fatton</td>
<td>Fatton</td>
<td>Node configuration</td>
<td>Emulab web interface, m</td>
<td>master host</td>
<td>Authorization structure</td>
<td>N/A</td>
<td>N/A</td>
<td>[3]</td>
</tr>
<tr>
<td>Flextab</td>
<td>Emulab container</td>
<td>Worker thread</td>
<td>Firefox add-on manager</td>
<td>Firefox browser tab</td>
<td>Package Fatton script</td>
<td>N/A</td>
<td>N/A</td>
<td>[22]</td>
</tr>
<tr>
<td>Grid5000</td>
<td>Physical machines and overlay network</td>
<td>OAR resource manager</td>
<td>Kademstore</td>
<td>REST API</td>
<td>Kademstore and Kademstore</td>
<td>OAR</td>
<td>Reference API</td>
<td>[20] [21]</td>
</tr>
<tr>
<td>NodeNet</td>
<td>Virtual edge node, core node</td>
<td>Assignment, Binding, Run phase</td>
<td>N/A</td>
<td>Create, Distillation, Phases</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>[22]</td>
</tr>
<tr>
<td>ORELLA</td>
<td>ORELLA</td>
<td>VT Planner</td>
<td>ODOT Control Framework and WebGUI</td>
<td>N/A</td>
<td>ODOT Control Framework &amp; WebGUI</td>
<td>N/A</td>
<td>N/A</td>
<td>[24] [25]</td>
</tr>
<tr>
<td>Planet Lab</td>
<td>VM slice</td>
<td>Node manager</td>
<td>Node manager, bootCD</td>
<td>Slice creation service, ssh</td>
<td>PlanetLab Central(TLC)</td>
<td>N/A</td>
<td>N/A</td>
<td>[20] [21]</td>
</tr>
<tr>
<td>RON</td>
<td>RON client, grid tit AIR</td>
<td>Forwarder object</td>
<td>N/A</td>
<td>N/A</td>
<td>RON router (impliments routing protocol)</td>
<td>N/A</td>
<td>RON membership manager, fuzzer</td>
<td>[1]</td>
</tr>
<tr>
<td>Satellite Lab</td>
<td>Satellite</td>
<td>N/A</td>
<td>N/A</td>
<td>Manual</td>
<td>OS-specific installers</td>
<td>Manual Allocation</td>
<td>Heartbeat mechanism</td>
<td>[1]</td>
</tr>
<tr>
<td>SPRINT</td>
<td>SPRINT</td>
<td>Daemon for sphy and libraries</td>
<td>N/A</td>
<td>Command line, Web interface</td>
<td>Jobs process</td>
<td>Controller (splayd)</td>
<td>Splay (keeps track of all daemons and processes)</td>
<td>[25]</td>
</tr>
<tr>
<td>VINI</td>
<td>PlanetLab (server, extended CPU scheduling, virtual network devices)</td>
<td>Same as PlanetLab (with extensions such as virtual point-to-point connectivity, distinct forwarding tables and routing protocols per virtual node, etc.)</td>
<td>N/A</td>
<td>N/A</td>
<td>VINI</td>
<td>N/A</td>
<td>[27]</td>
<td></td>
</tr>
</tbody>
</table>

Noteworthy: WOS3 routers running a custom version of OpenWrt distribution.

1 [An Experimentation Workbench for Repeatable Networking Research**, NSDI'07 [70].
2 The API that a RON client uses to send and receive packets.
3 A Lua-based sandbox.

[https://wiki.emulab.net/wiki/vnodes](https://wiki.emulab.net/wiki/vnodes)

[https://wiki.emulab.net/wiki/vnodes](https://wiki.emulab.net/wiki/vnodes)

[https://wiki.emulab.net/wiki/vnodes](https://wiki.emulab.net/wiki/vnodes)
students shows that students were able to create and deploy a custom private testbed and use it to run an experiment in under one hour. Our evaluation of five testbeds constructed with Tsumiki highlights some of the use cases: we reproduce previously published results on Sensibility Testbed, construct Ducky using Docker to sandbox experiment code, evaluate availability in one’s social network using the Social Compute Cloud, demonstrate the flexibility of testbed deployment on wireless routers through SOAR, and overview the research and educational impact of the Seattle testbed.

Acknowledgments. We would like to thank the 100+ people who have made improvements to the Tsumiki code base, particularly the Seattle project, and over 4K people who have used our testbeds. This work has been extensively supported by NSF, grant numbers 1547290, 1405904, 1405907, 1205415, 0834243, and 1223588. This work has also been supported by an NSERC Postdoctoral Fellowship.

References

Justin Cappos is an Associate Professor at NYU who focuses on solving systems and security problems in widely used software. His published research works are adopted into production use by widely used software including Docker, Python, VMware, several models of automobiles, Cloudflare, Digital Ocean, and popular Linux distributions.

Yanyan Zhuang is currently an Assistant Professor at the University of Colorado, Colorado Springs. Her research interests include networked systems, mobile and wireless networks, security and privacy, as well as software engineering. More information is available on her homepage: [http://www.cs.uccs.edu/~yzhuang/](http://www.cs.uccs.edu/~yzhuang/).

Albert Raftesder is a Research Professor at the Computer Science and Engineering Department at New York University, and the current technical lead for the Seattle Testbed project. He takes an interest in building useful, usable experimental platforms for research, education and the general public, and participatory aspects of computer networks.
Ivan Beschastnikh is an Assistant Professor in the Department of Computer Science at the University of British Columbia. He has broad research interests that touch on systems and software engineering. His recent projects span distributed systems, formal methods, modeling, and program analysis. More information is available on his homepage: http://www.cs.ubc.ca/~bestchai/