Tsumiki: A Meta-Platform for Building Your Own Testbed[*]

Justin Cappos[ny] Yanyan Zhuang[✤ny] Albert Rafetseder[ny] Ivan Beschastnikh[✤]
[ny] New York University, [✤] University of British Columbia

Contents

[*]This document provides supplementary materials for [19].

List of Figures

List of Tables

1 Introduction

Testbeds—such as RON [6], PlanetLab [44], Emulab [69], and GENI [31] —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 DHT research, while BISmark [62] and SamKnows [56, 63] have impacted Internet policy decisions at the 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 to deploying long-running Internet services across a wide area. However, PlanetLab gives a skewed view of Internet connectivity [8, 45, 61]. This deficiency prompted researchers to create new testbeds, such as SatelliteLab [27], BISmark [62, 63], and Dasu [57]. These are designed to 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 testbed software.

In this work we show that code reuse between testbeds does not contradict the construction of diverse testbeds. We formulate a set of testbed design principles and use these in concert with existing testbeds designs to propose a component-based model that we call Tsumiki[1]. Tsumiki is a meta-platform for testbed construction that makes it easy for researchers to build and customize new testbeds.

Tsumiki consists of seven components (Figure 1) that are configurable and replaceable. Tsumiki components are characterized by open, well-defined APIs that enable different component *realizations* (i.e., potentially diverse implementations of a component that use the same interfaces), to work together without strong dependencies. Although the interfaces are tightly specified, the components in Tsumiki are not rigidly defined — 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 in different ways, such as a Docker container or a programming language sandbox. As another example, the experiment manager component can be realized as a tool for automated deployment of long-running experiments [42], a parallel shell [58], or a GUI-based tool [46].

We used the Tsumiki principles in 2008 to design and build the Seattle testbed [14]. Seattle, and several other subsequent Tsumiki testbeds, were built as a community effort, with contributions from around 100 developers from 30 institutions world-wide. This is the first paper to describe testbeds built using Tsumiki, including the first technical description of the Seattle testbed.

This work makes the following contributions:

⋆ **We use four principles to derive a modular testbed design called Tsumiki.** This design introduces a set of components that can be used to quickly and easily construct new testbeds. Tsumiki's design provides testbed developers the flexibility to extend, replace, or omit components to meet their needs while allowing them to reuse existing functionality.

⋆ **We implement Tsumiki and demonstrate that it enables the construction of a diverse set of testbeds in practice.** Tsumiki has been used to build and deploy testbeds like the *Social compute cloud* [20, 22] (a testbed that provides experimenters with access to resources on their Facebook friends' devices), *ToMaTo* [40] (a network virtualization and emulation testbed), *Sensibility* [72] (a testbed that provides IRB-approved access to sensors on Android devices), and *Seattle* [14] (a testbed for networking, security, and distributed systems research and education). To further illustrate the supported diversity, we prototyped a new testbed called *Ducky*, which uses Docker containers for sandboxing experimenter code.

[1]Tsumiki means "building blocks" in Japanese.

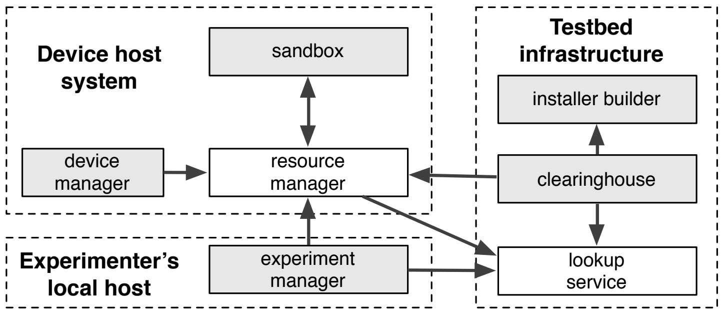


Figure 1: Components in Tsumiki are represented with boxes and the lines represent interfaces between the components. The arrows indicates the direction of the communication, with a bi-directional arrow indicating that calls may be made in both directions. The shaded components have an interface that is used by a human (not shown).

- ⋆ **We show that developing testbeds using Tsumiki is easy.** It took one of the authors (who had never used Docker) three hours to build the Ducky testbed. Ducky's construction only required adding or changing 66 lines of code in Tsumiki and required no changes to Docker. We also report on a user study with seven graduate students, in which each participant used Tsumiki to build a new testbed and then used it to deploy a small experiment. Overall, participants with little experience in using networked testbeds used Tsumiki to create a custom testbed, deployed it, and used it to run an experiment, all in under an hour.

- ⋆ **We show that testbeds constructed with Tsumiki are useful for research.** Tsumiki-based testbeds have been widely used for research and education. For example, the Seattle testbed serves updates to about 39K devices, including smartphones, tablets, and laptops. Over its six years of operation, over four thousand researchers have created and used a Seattle clearinghouse account. There have also been dozens of research projects that used Tsumiki-based testbeds [24, 29, 30, 38, 40, 65, 66, 68, 71]. We also show how the Sensibility testbed can be used to reproduce a study of WiFi rate adaptation algorithms [50]. This reproduction helped us uncover a mistake in the original paper's description of the algorithm.

2 Detailed Tsumiki design

Tsumiki's goal is to make it easy to reuse testbed functionality in new testbeds. It achieves this goal through a set of components and interfaces that can be used across a broad array of possible testbeds. Tsumiki has seven components, as shown in Figure 1. We first overview these components in Section 2.1. Then, in Section 2.2 we describe the key set of interfaces in Tsumiki. We introduce the main data structures used in Tsumiki in Section 2.3, and then in Section 2.4 we show some use scenarios where different components and interfaces work together.

2.1 Tsumiki components

The components in Tsumiki have three possible locations: the device host system (where an experiment runs), the testbed provider (that organizes and keeps track of testbed resources), and the experimenter's local host (where experiments are initiated).

2.1.1 Device host system

Devices provide resources for experimenters to use in their experiments. To isolate experimenter code from the device host system, experimenters' code executes in a **sandbox** component. The default sandbox in Tsumiki is a Python-based sandbox [15] that uses language-based isolation to mitigate the impact of bugs in experimenter code. The **resource manager** listens for remote commands and mediates access to the sandboxes on a device (authenticates access, starts and stops sandboxes, etc). Finally, the **device manager** handles installation and software updates of the sandbox and the resource manager. During installation the device manager optionally benchmarks the device resources. Once installed, it periodically checks in with the testbed provider and re-installs the local components when there are software updates.

Location	Component	Function
Device host system	Sandbox	Isolates experimenter code.
	Resource manager	Mediates access to sandboxes.
	Device manager	Installation and software updates.
Testbed provider	Clearinghouse	Tracks experimenters, mediates experimenter access to devices.
	Install builder	Builds custom installers.
	Lookup service	Device discovery.
Experimenter's local host	Experiment manager	Helps experimenters to deploy and run experiments.

Table 1: Tsumiki components, and their function and location.

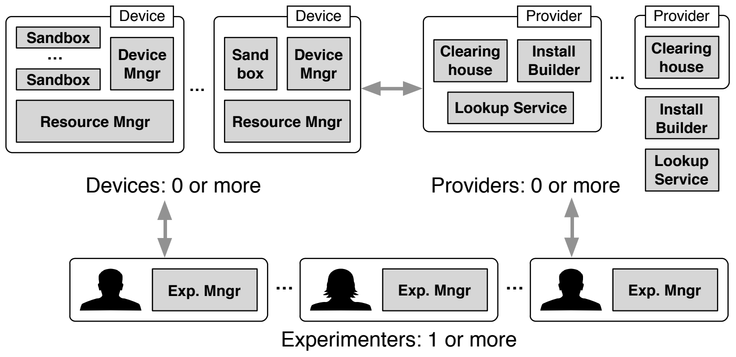


Figure 2: Tsumiki architecture and different components.

2.1.2 Testbed provider

The testbed provider helps experimenters to acquire and manage testbed resources, and may implement policies to incentivize participation in the testbed. A **clearinghouse** keeps track of experimenters and mediates experimenter access to the available sandboxes. Its key role is to facilitate device sharing, which relieves individual experimenters from recruiting devices to join their testbeds. The clearinghouse also allows device owners to download the Tsumiki software to add new devices to the testbed. To join Tsumiki, a device owner first accesses an **install builder** and downloads an installer for his host system. The device manager, resource manager, sandbox, and a set of public keys are packaged into this installer by the install builder. After an installation, the resource manager on the device uses the public keys to register the device in a **lookup service** so that the device can be later discovered by experimenters and other testbed components. Although we associate the install builder and lookup service with the testbed provider, in practice, these two components do not need to be under a provider's control. Furthermore, the Tsumiki model can support multiple providers that mediate access to the same set of devices. That is, a device may simultaneously participate in more than one testbed.

2.1.3 Experimenter's local host

Experimenters use their local machines to initiate and control experiments on a Tsumiki-based testbed. The **experiment manager** allows experimenters to deploy and run experiments in sandboxes on remote devices they requested through the clearinghouse. The experiment manager first looks up the set of sandboxes associated with an experimenter in the lookup service, and caches this list. It then communicates directly with the resource manager running on these devices, executing commands on behalf of the experimenter.

The seven components and their location are listed in Table 1. Note that there can be multiple sandboxes on the same device, and the install builder and lookup service do not need to be associated with a testbed provider. As mentioned above, the Tsumiki model can also support multiple providers that mediate access to the same set of devices. Therefore, the Tsumiki architecture in practise looks like Figure 2.

2.2 Tsumiki interfaces

In aspiring to reuse components across testbeds we had to strike a balance between interoperability and over-prescribing component behavior. We did this by defining the interfaces between components so that multiple *realizations*, or implementations, of a component were possible. Different component realizations use the same interface, but may be optimized for different situations or user communities. For example, there is a realization of experiment manager that supports automated deployment management [41] and another realization that provides a parallel shell for interactive management of experiments [58]. In this section we describe the *inter-component* and *human-visible* interfaces, and default realizations of the above mentioned components in further detail.

2.2.1 Lookup service

Inter-component interface. The lookup service component only has an inter-component interface: `get`/`put` calls to retrieve values associated with keys and to associate keys with values, respectively. This component is used by the resource manager to advertise device availability, and by the clearinghouse and the experiment manager for locating devices in the testbed. This is especially important for mobile devices, which frequently change their IP address.

The lookup service interface to the other components (e.g., resource manager, clearinghouse, and experiment manager) is as follows.

- `put(key, value)` is used by *resource manager* as: `put(admin_pubkey,ip:port)`, and `put(user_pubkey, ip:port)` for admins and users, respectively[2].
- `value = get(key)` is used by *clearinghouse*'s `check_new_install_daemon()` and `check_online_sandboxes_daemon()`, `ip:port = get(user_pubkey)` calls to look for newly installed sandboxes and online sandboxes. This interface is also used by *experiment manager*'s `browse` command to locate the available devices associated with the experimenter's key.

For increased fault tolerance and availability Tsumiki supports seven different realizations of the lookup service. These range from distributed hash tables, such as OpenDHT [52], to centralized variants that support TCP and UDP protocols, NAT traversal, and other features (Table 5 in Section 4). The installer builder bundles software that indicates the lookup services to be used by devices participating in the testbed.

2.2.2 Sandbox

The purpose of the sandbox component is to contain and execute experimenter code. It has both inter-component and human-visible interfaces.

Inter-component interface. The sandbox's inter-component interface allows the resource manager to control a hosted experimenter program. This interface has just four calls: `start()`, `stop()`, `status()`, `getlog()`.

Human-visible interface. The sandbox's human-visible interface is a programming interface to experimenter programs to access resources on the device. There are different instances of the sandbox component, which vary in details such as whether they execute binaries or code in a specific programming language. The *Repy* (Restricted Python) sandbox, a widely used Tsumiki sandbox, executes a subset of the Python language and provides a set of system calls available to experimenter code, as listed in Table 2. A detailed description of these calls can be found at `https://seattle.poly.edu/wiki/RepyV2API`.

Sandbox design criteria. The different instances of the sandbox component all follow three criteria: security isolation, performance isolation, and code portability. We illustrate these features by describing the Repy sandbox.

1. *Security isolation.* The sandbox component must be able to execute experiments, yet represent as little risk to the device owner as possible. The Repy sandbox uses language-based isolation and is implemented in Python. This sandbox uses *security layers* to push library functionality out of the sandbox kernel, thereby helping to mitigate the impact of bugs in libraries [15]. As a result, the sandbox maintains containment of application code despite bugs in the standard library implementation.
2. *Performance isolation* prevents interference between experimenters on shared devices. Without performance isolation, it is possible for a single experiment to transmit packets as fast as the hardware allows. The Repy sandbox is lightweight yet has similar performance isolation as OS VMs: the sandbox uses OS hooks to monitor

[2]The concept of admins and users is introduced in Section 2.2.3.

File system	**Threading**
`openfile`	`createlock`
`close`	`lock.acquire`
`readat`	`lock.release`
`writeat`	`createthread`
`listfiles`	`sleep`
`removefile`	`getthreadname`

Network	**Miscellaneous**
`gethostbyname`	`log`
`getmyip`	`getruntime`
`sendmessage`	`randombytex`
`openconnection`	`exitall`
`socket.close`	`createvirtualnamespace`
`socket.recv`	`virtualnamespace.evaluate`
`socket.send`	`getresources`
`listenforconnection`	
`tcpserversocket.getconnection`	
`tcpserversocket.close`	
`listenformessage`	
`udpserversocket.getmessage`	
`udpserversocket.close`	

Table 2: System calls in Tsumiki's Repy sandbox realization.

and control the amount of CPU and memory used by a sandboxed program. To restrict other resources, such as network and disk I/O, the sandbox interposes system calls that access these resources and prevents or delays the execution of these calls if they exceed their configured quota [37].

3. *Portability* of experimenter code allows experimenters to leverage a wide array of devices without needing to port their programs. The sandbox exposes a programming interface that is portable across diverse device types [49]. This interface provides methods to access the network through UDP and TCP sockets, to read and write files in a dedicated folder, and utility methods to retrieve the current time, etc.

Sandbox porting experience. The core Repy sandbox implementation was written to resemble POSIX. As a result, the initial version of the Repy sandbox mostly worked on Windows, Linux, Mac OS X, and BSD. If a platform is fairly similar to an existing platform that is supported, the development effort is minor. For example, it took one developer a week of effort to port the Repy sandbox to the Nokia N800 since the Nokia N800 is similar to Linux. Following this, researchers at Nokia's research lab were able to port the sandbox to the Nokia N900 in a few days. Similarly, researchers at the University of Vienna were able to do early stage ports of the Repy sandbox to Android and jailbroken iPad / iPhone / iPod devices with a few weeks of developer effort.

2.2.3 Resource manager

A device must not only be shared by experimenters, it can also participate in multiple testbeds which may not collaborate with or trust each other. To facilitate shared control of devices across experimenters and testbeds, Tsumiki uses a resource manager component. As the resource manager is not directly visible to an experimenter or device owner, it has only an inter-component interface like the lookup service.

Inter-component interface. The resource manager uses the concept of keys to mediate access to the sandboxes on a device. Each sandbox has one admin and zero or more users. Therefore, every Tsumiki sandbox has two kinds of keys controlled by the resource manager: a set of *user keys* and one *admin key*. A user key grants the key holder access to run code in a sandbox. For example, an experimenter uses his/her user key to run experiments. The admin key grants the key holder the privilege to perform actions such as changing the set of user keys, and allocating resources between the sandboxes on a device. The resource manager exposes three categories of calls, grouped by admin or user permission [2].

Calls private to admins. The first category is the calls private to admins, which include calls to change admin/user information, and calls to split/join sandboxes that allows the resource manager to control the allocation of resources on the device among the sandboxes.

1. Admin/user operations:

- `ChangeAdmin(sandboxname, newpublickey)` allows the admin (typically the *clearinghouse*) to set a new public key for the admin. `sandboxname` is a string to indicate a unique sandbox on a device. This call changes the admin key of a sandbox to `newpublickey`, and also resets the admin's information string[3] to an empty string. This can be used to transfer devices between clearinghouses, or testbeds.

  For example, this call is used in clearinghouse's `check_new_install_daemon()` to set a unique key for admin.
- `ChangeUsers(sandboxname, listofpublickeys)` is used by the *clearinghouse* to grant and revoke experimenter access to a sandbox (e.g., when an experimenter requests resources). This allows for multiple experimenters to have access to the same sandbox. This call changes the list of public keys selected for the sandbox (`listofpublickeys` may only have a small number of entries).

  For example, this call is used in clearinghouse's `acquire_resources(auth,rspec)`, `release_resources(auth, list_of_handles)`, `acquire_specific_sandboxes(auth,sandboxhandle_list)` calls to set experimenter permission on the acquired sandbox.
- `ChangeAdminInformation(sandboxname, informationstring)` sets the admin's information string to a specific value. For example, it is used by clearinghouse in `check_new_install_daemon()` to track information about a sandbox (by using a special label in `informationstring`).

2. Split/join sandboxes, allows the resource manager to split a sandbox into smaller ones, or join small sandboxes into a big one, via controlling the allocation of resources on the device:

- `SplitSandbox(sandboxname, resourcedata)` splits a sandbox into two smaller sandboxes. `resourcedata` determines the size of one of the new sandboxes. Both sandboxes are considered new and are given new `sandboxname`'s. The admin key is copied from the existing sandbox. The restrictions are all removed and must be re-added by the admin. The `sandboxname`, filesystems and logs of the sandboxes are newly created. This call returns the names of the new sandboxes. Splitting also causes offcut resources, i.e., the cost of splitting.

  For example, this call is used by clearinghouse in `check_new_install_daemon()` to divide resources between experimenters when new installs are found.
- `JoinSandboxes(sandboxname1, sandboxname2)` merges two sandboxes into one larger sandbox. The resource information is determined by adding the resources in the two sandboxes along with the offcut resources. A new sandbox is created with a new `sandboxname`. The restrictions are written so that any action that either could have performed can be performed by the created sandbox. The admin key must have previously been identical on both sandboxes and will be copied to the new sandbox. The filesystem and log of the new sandbox is freshly made.

  For example, this call is used by clearinghouse in `check_new_install_daemon()` to collect and re-combine expired or freed resources to one bigger, free-to-assign/split sandbox.

Calls private to admins and users. This class of resource manager calls manipulate the program and state inside a sandbox. For each sandbox on the device the resource manager maintains a state machine shown in Figure 3. It regularly checks the status of a sandbox and updates the corresponding state machine. 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 resource manager loses the ability to communicate with the sandbox, then it updates the sandbox state to `Stale`. `Stale` can recover to its previous state (dashed transitions in Figure 3). Finally, the sandbox can be reset in any state back to `Fresh` through the `ResetSandbox()` resource manager call. The complete list of calls with arguments are as follows.

[3]The admin's information string, `admininformation`, contains opaque data about the sandbox that the admin defines. This information is flushed anytime the sandbox's admin key changes. The information is a field that only the current admin could have set and may be used to advertise a service, etc.

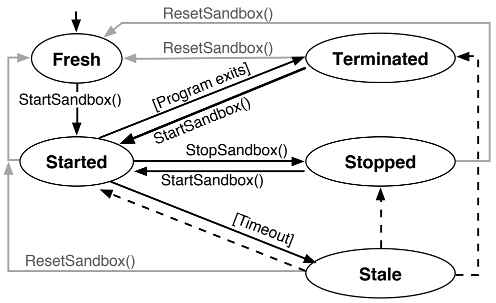


Figure 3: The sandbox state machine maintained by the resource manager. `StartSandbox`, `StopSandbox`, and `ResetSandbox` are resource manager calls. [Program exits] and [Timeout] are sandbox events. Dashed arrows indicate an implicit event in which the sandbox recovers after a timeout.

- `StartSandbox(sandboxname,program_platform,args)` begins running a sandbox with a set of arguments (including the command name) on a given programming platform (RepyV1/RepyV2/Docker, or any other supported sandbox type). For example, it can be used by experiment manager to start an experiment (e.g., when experiment manager calls `run program [args]`).
- `StopSandbox(sandboxname)` stops the execution of a sandbox. For example, it can be used by experiment manager to stop an experiment (e.g., when experiment manager calls `stop`).
- `ResetSandbox(sandboxname)` removes all files in a sandbox's file system, resets the log, and stops the sandbox if it's running. For example, it can be used by the experiment manager to reset an experiment (e.g., when experiment manager calls `reset`). It can also be used by clearinghouse in call `release_resources(auth, list_of_handles)`, when an experimenter releases a sandbox (or it expires) to reset the sandbox's file system, logs, etc.; also used in clearinghouse's `check_new_install_daemon()` call to reset the sandbox when necessary.

In addition to calls that manipulate the experimenter's program inside the sandbox, the resource manager exposes calls to manipulate the file system inside the sandbox (add, remove, list, download files) and a call to retrieve the sandbox console log. These calls are listed as follows.

- `AddFileToSandbox(sandboxname, filename, filedata)` creates (overwrites if it exists) a file `filename` in the sandbox with contents `filedata`. This operation can fail if the file system of the sandbox is too small.
- `RetrieveFileFromSandbox(sandboxname, filename)`.
- `DeleteFileInSandbox(sandboxname, filename)`.
- `ListFilesInSandbox(sandboxname)`.

  The above calls can be used by an experiment manager, e.g., when an experiment manager calls `upload filename`, `download filename`, `show files`.
- `ReadSandboxLog(sandboxname)` returns the sandbox's console log. It can be used by an experiment manager to read the log from an experiment (for example, when experiment manager calls `show log`).

Public calls. These calls allow anyone to find out the status of a sandbox (`GetSandboxStatus()`) and to determine the resources associated with a sandbox (`GetSandboxResources()`).

- `GetSandboxStatus()` returns the sandbox name, admin key, status, user key(s), and admin information for all sandboxes on a local machine. It can be used by an experiment manager to get the status of all sandboxes, e.g., when an experiment manager calls `browse` and `list`. It can also be used by clearinghouse in `check_online_sandboxes_daemon()` to know about the status of a sandbox.
- `GetSandboxResources(sandboxname)` can be used by clearinghouse in `check_new_install_daemon()` to determine the resources associated with a sandbox. Similarly it can be used by an experimenter through experiment manager, for example, when experiment manager calls `show resource`.

As the resource manager needs to run on a wide variety of devices, it must be portable. Rather than us solving the portability issue for the sandbox, resource manager, and device manager separately, we consolidate the code. The resource manager leverages a portability library that is exposed by our core sandbox. Thus the resource manager and device manager are portable, modulo the parts that make calls not covered by the portability library (such as calling a Python library to perform a non-portable OS operation).

2.2.4 Device manager

Inter-component interface. Besides running experimenter code, a device in a testbed needs to periodically install updates to the sandbox and the resource manager. As well, there must be a way for the device owner to control resource allocation (e.g., to limit the percent of the CPU that testbed code is allowed to use). The device manager includes scripts to stop and start the Tsumiki software on the device, a software updater, and a program that the device owner can use to control the network interfaces and resources allocated. Since the device manager's human-visible interfaces can vary significantly, we only describe its inter-component interface here.

The list of calls of the device manager inter-component interface are as follows.

- `StopResourceManager()` is used to stop the resource manager, particularly when there is an update available (the resource manager must be turned off in order to update the software).
- `StartResourceManager()` is used to during installation by the device manager to benchmark the system resources (only performed once). It generates a resource file to be used by resource manager's `GetSandboxResources()` call.
- `AutoUpdate()` runs in background, sleeps some random amount of time (30min - 1hour) to check with the install builder if there exists a new update. It then downloads the new files for the installer from install builder and replaces the old installer.

2.2.5 Clearinghouse

It is helpful for experimenters to share a pool of devices. A natural way of coordinating this sharing is through a clearinghouse site that allocates resources according to some policy. While a clearinghouse is not used in many cases, it is needed for a testbed to scale. A clearinghouse allows experimenters to register accounts and share access to a common pool of sandboxes.

Human-visible interface. Since clearinghouses' human-visible interfaces can vary significantly across different testbeds, we only describe the interface of a widely used Tsumiki clearinghouse, the Seattle clearinghouse [5]. The interface calls of this clearinghouse are as follows.

- `acquire_resources(auth,resource_specification)`. Given a resource specification `resource_specification`, this call acquires resources for the experimenter, identified by an authentication structure `auth`, which takes the form of a dictionary: `{'username':'YOUR-USERNAME','api-key':'YOUR-API-KEY'}`.

  There are currently four types of `resource_specification` defined: LAN (sandbox on nodes with the same starting three octets of the IP address), WAN (sandbox on nodes with different starting three octets of the IP address), NAT (sandbox on nodes that communicate through a NAT forwarder), and random (a combination of the above).

  Depending on the implementation, this call can be used by an experiment manager to acquire sandboxes (for example, when experiment manager calls `get 3 wan`).
- `acquire_specific_sandboxes(auth,sandboxhandle_list)` acquires specific sandboxes (sandboxes of specific names on specific nodes). This will be best effort. Zero or more sandboxes may be acquired. Requesting sandboxes that exist but are not available will not result in an exception. `sandboxhandle_list` is a list of sandbox handles. This call returns a list of dictionaries like that returned by `acquire_resources()`.
- `release_resources(auth,list_of_handles)` releases resources (sandboxes) previously acquired by the experimenter. The handles in `list_of_handles` indicate the sandboxes to be released. Used by an experiment manager to release sandboxes (for example, when an experiment manager calls `release browsegood`, where `browsegood` is the name of a group of sandboxes).

- `renew_resources(auth,list_of_handles)` renews resources (sandboxes) previously acquired by the experimenter. The handles in `list_of_handles` indicate the sandboxes to be renewed. By default sandboxes are renewed for 7 days.
- `get_resource_info(auth)` returns a list of resources (sandboxes) currently acquired by the experimenter. The return list is of the same form as returned by `acquire_resources()`.
- `request_installer()` is the API equivalent of a device owner clicking a link to request an installer for his/her device. When receiving this request, the clearinghouse asks the installer builder to build an installer that includes the clearinghouses keys, and returns the generated URL for download.

The clearinghouse also runs the following daemons in the background:

- `check_new_install_daemon()` looks for new installs via lookup service's `get()` call, then uses resource manager's `ResetSandbox()`, `SplitSandbox()`, `JoinSandboxes()` calls to set up the resources to be ready to hand out.
- `check_online_sandboxes_daemon()` looks for sandboxes that are currently online periodically, via the `get()` call in the lookup service and the `GetSandboxStatus()` call in the resource manager (check if the device is contactable, as non-transitive connectivity or other network errors can cause a device to go down or offline).

Clearinghouses vary widely in their policies. For example, in the social network testbed (Section 7), the clearinghouse would query a social network, like Facebook, to determine which "friend" resources are available to an experimenter. A clearinghouse facilitates device sharing; without a clearinghouse each experimenter would need to individually recruit devices to join their testbed.

2.2.6 Install builder

The device manager, resource manager, and sandbox are packaged together into an installer by the install builder. A device owner can access the install builder's server and download an installer that has all of these components. This installer is customized to include the appropriate user and admin keys[4].

Inter-component interface. Different groups will frequently set up a new custom installer builder component if they wish to utilize different versions of the components, such as a unique sandbox. An installer builder only has an inter-component interface. The list of installer builder calls include:

- `api_version()` returns the current API version of the install builder as a string. It is often used by clearinghouse's `request_installer()` call to get the latest API version.
- `build_installers(sandboxes,user_data)` builds an installer for each of the platforms the install builder supports building for, using the given sandboxes and user information. In the absence of provided user data, cryptographic keys will automatically be generated. It is often used by clearinghouse's `request_installer()` call to build a new installer for a specific request.
- `get_urls(build_id)` returns a dictionary containing the URLs from which installers for the given build ID may be downloaded. It is often used by clearinghouse's `request_installer()` call to generate a URL for the newly built installer.

2.2.7 Experiment manager

In a distributed testbed, an experimenter must deploy and run code across multiple remote devices. Tsumiki provides experimenters with an experiment manager that experimenters run locally to communicate with resource managers and their remotely deployed experiments. Since an experiment manager is directly human-visible, we describe its human-visible interface here.

Human-visible interface. There are variations of such tools that service different user bases and serve different roles, such as tools for automated deployment of long-running experiments. The most widely used tool is an interactive shell called `seash`[5]. Using `seash`, an experimenter can communicate with remote devices, upload and run experiments, collect experiment results, and do all of these in parallel.

[4]As an example, here is an installer created by the install builder in the Seattle testbed: `https://seattleclearinghouse.poly.edu/download/flibble/`

[5]`https://seattle.poly.edu/wiki/SeattleShell`

Experiment manager command	Corresponding resource manager calls
`run program-name [args]`	`AddFileToSandbox(program-name, ..)` `StartSandbox(program-name, args, ..)`
`stop`	`StopSandbox()`
`list`	`GetSandboxStatus()`
`show log`	`ReadSandboxLog()`
`download filename`	`RetrieveFileFromSandbox(filename, ..)`
`upload filename`	`AddFileToSandbox(filename, ..)`
`show files`	`ListFilesInSandbox()`
`browse`	Lookup service: `Get(user-pubkey)` Resource manager: `GetSandboxStatus()`

Table 3: Commands for the default experiment manager in Tsumiki and the corresponding resource manager calls; `browse` is an exception (it makes calls to both the lookup service and the resource manager). The experiment manager commands are issued by an experimenter from his local host.

The common seash calls work as follows. First, an experimenter uses an experiment manager command on his local host to load his testbed credentials so that his local experiment manager looks up the available devices under his control through the lookup service. Next, the experimenter issues experiment manager commands to directly communicate with the resource manager on remote devices which controls the sandbox. For example, to run a `helloworld` program the experimenter issues the `run helloworld` command from the experiment manager, which instructs the remote resource managers to upload the file `helloworld` from his local host to the sandbox, and then start executing this program. The experimenter can then check the state of his sandbox, and look through the console log output of the `helloworld` program.

A set of common experiment manager commands and the associated (remote) resource manager calls are listed in Table 3. The common `seash` calls are as follows.

- `loadkeys username` loads an experimenter's (registered with `username`) public and private keys.
- `as username` means you are now this experimenter.
- `browse`, as listed in Table 3, browses the set of resources available to this experimenter. It first uses lookup service's `get()` call to find the list of `IP:port` of the hosts with sandboxes available to this experimenter. It then uses resource manager's `GetSandboxStatus()` call to get the sandbox names and displays them to the experimenter.
- `on groupname` selects every sandbox in `groupname` as the target for the following operations.
- `show ip` prints the IP address of all current target sandboxes.
- `show location` prints the city, country of all current target sandboxes via a geoip service.
- `list` prints the status of all current target sandboxes. It uses resource manager's `GetSandboxStatus()` call to get the status (`Fresh`, `Started`, `Terminated`, `Stopped`, `Stale` in Figure 3) of the sandboxes.
- `upload filename` uploads a file to all current target sandboxes. It uses resource manager's `AddFileToSandbox()` call.
- `download filename` downloads a file from all current target sandboxes. It uses resource manager's `RetrieveFile -FromSandbox()` call.
- `show files` prints all the files uploaded to the current target sandboxes. It uses resource manager's `ListFilesIn -Sandbox()` call.
- `show log` prints console log form all current target sandboxes. It uses resource manager's `ReadSandboxLog()` call.
- `start program [args]` starts running program on all current target sandboxes. It uses resource manager's `Start -Sandbox()` call.
- `run program [args]` first uploads the program to all current target sandboxes, and then starts running the program. It uses resource manager's `AddFileToSandbox()` and `StartSandbox()` calls.
- `stop` stops running program on all current target sandboxes. It uses resource manager's `StopSandbox()` call.

- `reset` resets all current target sandboxes. It uses resource manager's `ResetSandbox()` call.
- `get number type` acquires a number of sandboxes of a certain type. Type can be `lan`, `wan`, `nat`, or omitted. It uses clearinghouse's `acquire_resources()` call.
- `release groupname` releases the group of sandboxes on `groupname`. It uses clearinghouse's `release_resources()` call.

2.3 Tsumiki data structures

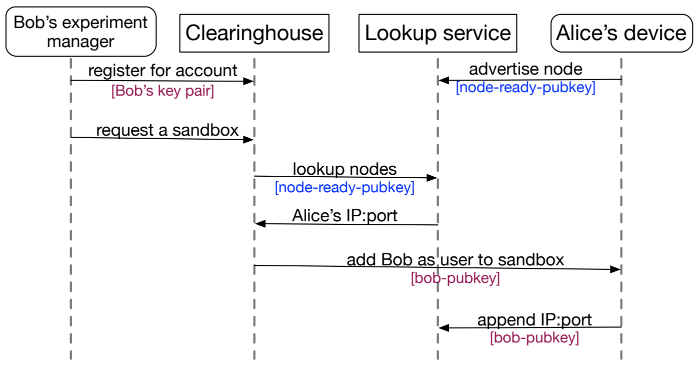
Tsumiki uses several data structures for maintaining information about devices, users, and other aspects of a testbed. Table 4 lists the most important data structures.

Location	**Data structure**	**Purpose**
Device host system	`benchmark results`	The amount of resources may be used by sandboxes; created during benchmarking.
Device host system, in installer package	`sandboxinfo`	Contains user/admin keys and relative sandbox sizes. Used during installation to create sandboxes (in conjunction with system benchmark values).
Device Manager directory	`sandboxdict`	Stores current user/admin keys, restrictions/log/stop files, sandbox status.
	`nodeman.cfg`	Node cryptographic key pair, various flags from installation (such as allowed devices), listen port for Device Manager
	restrictions file	Contains resource restrictions for a sandbox, conforming to `benchmark results`, `sandboxinfo`, and any join/split operations performed.
	Device Manager lock files	Used to ensure that only one of either processes is running; signals that the processes should terminate and restart following software updates.
Installing device's crontab or Registry	crontab / Windows Registry	Handles the restart of Device Manager and software updater following a restart.
`service sandbox` (a small sandbox with its own subdir)	Device Manager log files	Contain log output informing about important events such as restarts.
Device host system, in installer package; updated version on install builder site	software updater meta-info file	Contains cryptographic signatures of files in the installer. Software updater has the install builder's public key and URL, and can check for new metainfo files and correct signatures.
Device Manager directory	Sandbox directory	Only accessible place in the filesystem for sandboxes that run on the host.
Device Manager directory (configurable)	Sandbox log	Indirectly accessible logging space for sandboxes.
Clearinghouse	Clearinghouse node database	Data about any device where an installer was installed: keys, last known IP and resource manager port, software version, date added, date last contacted (for setting up sandboxes for experimenters), current status (online/healthy).
	Clearinghouse sandbox database	Node the sandbox is running on, sandbox name, experimenter who acquired it, date acquired, date of expiry, date added to database.
	Clearinghouse experimenter database	Contains data about experimenters (username, donor key, possibly user key, contact details, assigned resource manger port, date added), sandbox donations (count of active installations containing the experimenter's key), and sandbox acquisition.
Installer builder	Built installers cache	Makes it easy for experimenters to hand out installers (share link, not tarball).
	Software update website	Contains up-to-date metainfo file, and current version of files to be pushed during an update.
Experiment manager	Experiment session data	Any local data the experiment manager needs.
Lookup servers	key-value store	Dictionary mapping advertised keys to values.

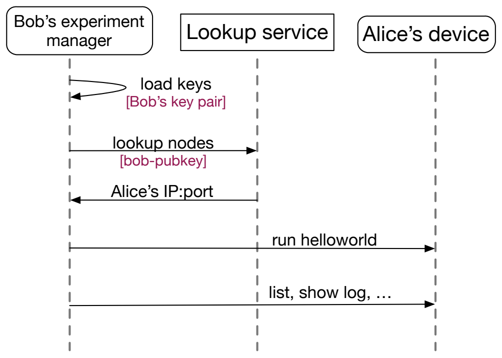
Table 4: The important data structures used in Tsumiki.

Every component maintains a set of data structures relevant to its function. Interacting components (Figure 2) synchronize their local state on demand through direct communication. Components co-located on the same host exchange data using the local filesystem. An example of this is the information in the *benchmark results* in Table 4, which is passed to the *restrictions files* used by the device manager. State that is exchanged between components on different hosts is serialized, encrypted, and cryptographically signed.

When distributed components cannot establish direct communication, state is propagated through the lookup service either directly as a *(handle, state)* tuple, or indirectly as *(handle, locator)*. An example of direct state transfer through the lookup service is when an experiment manager looks up a list of node identifiers (state) advertised under a certain experimenter's public key (handle). The state returned by this lookup represents an excerpt of the credentials (`sandboxdict` in Table 4) currently configured for a number of sandboxes on the network. This eliminates the



(a) Bob requests resources on a testbed device.



(b) Bob runs code in a device sandbox.

Figure 4: Bob runs an experiment.

overhead of querying each sandbox separately for the public key. An example of indirect state transfer through the lookup service is when the experiment manager first looks up the experimenter's public key and retrieves the device identifiers. Then, the device identifiers are used to contact each device and query it for the desired resource data.

2.4 Overview through testbed scenarios

We now walk through two testbed scenarios to show how different components in Tsumiki interact, and how trust boundaries (from principle P1) are practically realized. Figure 2 overviews how independent roles are associated with the different parts of the Tsumiki architecture.

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

Scenario 1: Bob runs an experiment. *Experimenter requests resources on a testbed device.* When Bob, an experimenter, registers for an account with the clearinghouse, he provides a public key, `bob-pubkey`, such that Bob has the corresponding private key `bob-privkey`. This key pair is unique to Bob. When Bob later requests a sandbox, the clearinghouse first uses the lookup service to lookup the available sandboxes. These sandboxes advertise themselves using the `node-ready-pubkey`, a key that is unique to the testbed. After finding a suitable sandbox, which satisfies the clearinghouse policy and Bob's constraints (e.g., only sandboxes on devices in North America), the clearinghouse contacts the resource manager running on the device. The clearinghouse instructs the resource manager to change the user of the sandbox on the device to Bob, by calling `ChangeUsers()`, which will set the user key for this sandbox to `bob-pubkey`. The resource manager then issues a `Put(bob-pubkey, IP:port)` call to append the IP and port of the resource manager to the current value of Bob's `bob-pubkey` in the lookup service. This allows Bob to look up the sandbox that he requested. This scenario is shown in Figure 4(a).

Experimenter runs code in a device sandbox. Next, Bob uses the experiment manager to control the acquired sandbox from his local machine. The default Tsumiki experiment manager is a shell through which experimenters execute commands (first column in Table 3). Bob first loads his public and private keys into the experiment manager. The experiment manager uses Bob's public key to look up the previously acquired sandbox in the lookup service with `Get(bob-pubkey)`. This returns the `IP:port` of the resource manager on the device. Bob can now make calls to the resource manager (through experiment manager commands) to control his sandbox. To run his `helloworld` program Bob issues the `run helloworld` command, which instructs the remote resource managers to upload the file `helloworld` from Bob's local host to the sandbox (by calling `AddFileToSandbox()`), and then start executing this program (by calling `StartSandbox()`). Bob can then check the state of his sandbox with the experiment manager `list` command, and look through the console log output of the `helloworld` program with the `show log` command. A set of common experiment manager commands and the associated (remote) resource manager calls are listed in Table 3. This scenario is shown in Figure 4(b).

Experimenter releases the acquired resources. When Bob is done with his experiment, he may release his sandbox

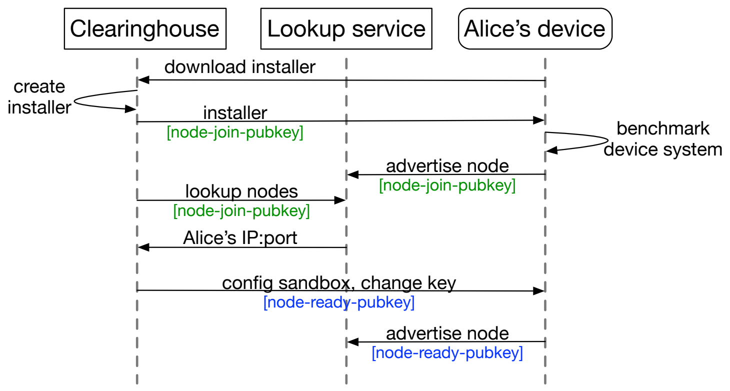


Figure 5: Alice contributes a device to the testbed.

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 will issue a `ChangeUsers()` call on the resource manager to remove `bob-pubkey` from the sandbox, and reset the sandbox (e.g., delete all files uploaded by Bob) with `ResetSandbox()`.

Scenario 2: Alice contributes a device to the testbed. In this scenario we re-use the sandbox user key. However, when Alice installs testbed software on her device, the sandboxes on her device will use a public key owned by a clearinghouse, not Alice, or another experimenter. Alice's device will advertise itself with this key to indicate its intent to join the testbed. This scenario is shown in Figure 5.

Device installation. When Alice, a device owner, decides to contribute her device to a testbed, she first downloads a testbed-specific installer through the clearinghouse. The clearinghouse creates this installer through the install builder component with the `BuildInstaller(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 install builder packages this key with the device manager, resource manager, and sandbox into a set of platform-specific installers. During installation, the device manager benchmarks the device system, and saves the result to use for later configuration.

Device discovery. There are three steps to discovering Alice's new device. (1) Since the installer included `node-join-pubkey`, the resource manager on the device has `node-join-pubkey` in a user sandbox after installation. When the resource manager starts, it advertises its `IP:port` of the sandboxes 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. (2) 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. (3) The clearinghouse asks the resource manager to change the `node-join-pubkey` on Alice's device to the `node-ready-pubkey` with `ChangeUsers()`. The resource manager will then advertise its `IP:port` using this new key and the clearinghouse will record that Alice's device successfully joined the testbed.

3 Customizing testbeds with policies

The previous section describes Tsumiki mechanisms: components and their interfaces. Additionally, Tsumiki encourages testbed developers to allow a component's behavior to be customized. This is achieved through policies that Tsumiki can enforce in three components: the clearinghouse, the resource manager, and the sandbox. For example, the resource manager policies are enforced by the resource manager that restricts the actions of a sandbox, depending on the different configurations.

We now describe a few policies, e.g., those can be used to incentivize participation in the testbed and determine the assignment of sandboxes to experimenters.

3.1 Clearinghouse policies

A sandbox admin, typically the clearinghouse, can affect the resource allocation among experimenters and sandboxes. The sandbox admin can also use appropriate API and restrict available resources for the sandbox.

Incentivizing participation. Testbed providers can implement clearinghouse policies to incentivize device owners to participate in the testbed. For example, the Seattle testbed (described in Section 8) uses a policy in which 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.

Matching experimenters with resources in their social network. The Social compute cloud testbed (described in Section 7) integrates with social network platforms, such as Facebook, and includes policies to match experimenters with resources provided by people in their social network.

Resource expiration. Experimenters may want to hold onto resources that they are not using. A simple policy to discourage this is to expire resource after a certain time period (i.e., the experimenter must re-acquire the resources every so often).

Resources allocation between sandboxes. The default policy in Tsumiki is to partition device resources equally between the sandboxes on a device. This is achieved by calling the resource manager's `SplitSandbox()` and `Join-Sandboxes()` calls (Section 2.2.3). However, this policy could be changed to, e.g., allocate fewer resources to sandboxes that host long-running experiments, or grant more resources to sandboxes controlled by a specific set of experimenters.

3.2 Resource manager policies

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 resource manager to apply these policies to control the sandboxes.

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.

Supporting multiple providers/testbeds. By default, a single device is associated with one provider. However, Tsumiki's mechanism can support the case where multiple providers mediate access to the same set of devices. A resource manager can allow a device owner to contribute his/her device resources to two distinct testbeds. Using the testbed scenario in Section 2.4, Alice's device would advertise itself using two different testbed-specific keys, namely, $\texttt{node\_join\_key}_{\texttt{A}}$ and $\texttt{node\_join\_key}_{\texttt{B}}$, to indicate that Alice wants her device to join both $\text{Testbed}_{\texttt{A}}$ and $\text{Testbed}_{\texttt{B}}$.

3.3 Sandbox policies

A sandbox user can control the resources the experiment code can have access to. The implementation of most sandbox policies relies on a technique called Security Layers [15]: API calls are transparently wrapped so that their capabilities can be restricted, while the function signatures and call semantics are unchanged.

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. A combination of these policies will result in an intersecting set of addresses and ports allowed. In a practical deployment, this could mean that a device owner blacklists their LAN, while an experimenter partitions 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 to the city the device is in [17,48].

Component Type	Project Name	Status	Developers
Sandbox	Repy V1	Production	J. Cappos, A. Dadgar
	Repy V2	Alpha	C. Meyer, J. Cappos
	Sensibility	Alpha	A. Rafetseder, Y. Zhuang, J. Cappos
	Lind	Devel	C. Matthews, S. Piyanan
	ToMaTo	Production	D. Schwerdel
	Ducky	Alpha	J. Cappos
	RepyV2 for OpenWrt	Alpha	X. Huang
Resource manager	Resource Manager	Production	J. Cappos, C. Barsan
Device manager	Seattle device manager	Production	J. Cappos, G. Condra
	Seattle-on-Android device manager	Production	Á. Lukovics, A. Rafetseder
	Sensibility device manager	Alpha	Á. Lukovics, A. Rafetseder
Experiment manager	Seash	Production	J. Cappos, A. Loh
	Overlord	Production	A. Loh, J. Samuel
	HuXiang	Beta	A. Loh
	Try Repy!	Beta	L. Pühringer, A. Rafetseder
	SeaStorm	Production	J. Kallin
	Owl	Production	S. Baker
Installer builder	Installer builder	Production	A. Hanson
Clearinghouse	Seattle clearinghouse	Production	J. Samuel, S. Ren
	Social clearinghouse	Production	K. Chard, S. Caton
	Sensibility clearinghouse	Devel	S. Awwad, K. Borra, A. Rafetseder, Y. Zhuang
	SeleXor	Devel	L. Law
Lookup service	OpenDHT	Production	S. Morgan, J. Cappos
	Digital Object Registry	Production	G. Manepalli, L. Lannom
	Centralized lookup service	Production	S. Rhea, A. Krishnamurthy
	UDP lookup service	Production	S. Morgan
	Repy V1 lookup service	Production	J. Cappos, C. Barsan, J. Samuel, L. Law
	Repy V2 lookup service	Production	S. Morgan, L. Law
	Secure lookup service	Devel	S. Morgan

Table 5: Tsumiki component implementation status and the main developers.

4 Implementation status

This section presents a wide array of customizations that we and the wider community of contributors implemented using the Tsumiki components. A listing of these customizations are listed in Table 5.

4.1 Core components

This subsection describes the implementation status of the Tsumiki core components derived in Section 2.1. We limit the discussion to those components that Sections 2.1 and 2.2 have not yet talked about.

4.1.1 Sandbox

Repy V1. The default sandbox in Tsumiki is a Python-based programming language sandbox. The Python subset used by the sandbox is called Repy, or Restricted Python [51]. It is executed by the Python interpreter and supports many built-in items from the standard Python implementation. We limit the subset of Python available in Repy by deliberately excluding such operations as `import`, `exec`, and `eval` [47] because the safety of these types of calls is difficult to verify. In order to minimize the risk of bugs, our sandbox implementation attempts to use only parts of the underlying trusted computing base that are stable, conceptually simple and widely used. For example, we allow use of classic Python classes (the overwhelmingly common use) as well as only simple types; we do not allow classes that subclass basic types, provide their own namespace storage mechanisms, or utilize other rarely used complexities that are new to the language.

We replace the built-in Python functions that write data to disk, utilize network bandwidth, or perform other complex tasks. To handle these tasks, we provide a high-level, resource metered API containing 17 API calls with another 14 calls accessible through objects. This provides programmers with the ability to read and write files on the disk, start threads, obtain locks, and send TCP and UDP traffic, along with other things [51]. We have found that by building the API with simple abstractions it is easier to reason about security, and prevents exploitation of low-level bugs.

Repy V2. Since our initial deployment of Repy V1, we have planned to provide better support from several aspects. First, we wanted to allow the experimenters to add security policies to all code that executes on their sandbox [15]. For example, the experimenter may want to restrict the code on devices using a policy that allows the device to only communicate with other devices under the same experimenter's control [9]. We extended the sandbox to support safe execution of policy code.

Another feature addition was to support the extensibility of the sandbox. Repy V2 can be extended with additional functionality so that it can utilize other devices, if the device owner allows it. For example, a researcher added support for accessing `tun/tap` interfaces from Repy V2 to support a project called **ToMaTo** [40,64]. These hooks are available on any devices that have enabled it.

Sensibility sandbox. Neither version of the above Repy sandbox includes calls to access sensors[6] on mobile devices per default. We extended the Repy V2 sandbox to allow access to sensor data. The details have been described in [19].

Lind. Ongoing work at the University of Victoria is extending Repy V2 to support code other than Python. The researcher will be able to compile their programs using the tool chain from Google Native Client [70] to validate the safety of code. The resulting code will execute computationally in the NaCl sandbox, but call into Repy V2 to perform system calls. This extension will give researchers the ability to execute anything that can be compiled for the x86 architecture and possibly other architectures such as ARM. This will be useful both for reusing legacy code and for writing performance critical experiments.

Ducky, as described in [19], is a Docker container based sandbox. The Ducky sandbox is implemented by setting up a new sandbox type in Repy, making the sandbox call Docker commands with appropriate arguments, and changing the default logging mechanism. The developer of Ducky used three hours to build this new sandbox.

RepyV2 for OpenWrt, described in [33], extends the RepyV2 API to allow for experimentation on home user WiFi routers, enabling broadband and wireless network experimentation. The project targets the free OpenWrt [3] embedded operating system, and allows Repy code to access statistic data of the network subsystem (such as interface byte counts and associated wireless stations) while maintaining experimenter code sandboxed.

4.1.2 Resource manager

Tsumiki's resource manager mediates resource access among sandboxes and provides a remote control interface into the sandboxes. It determines which experimenters may have access to a sandbox. Cryptographically signed communication is used to perform authentication of remote experimenters who are identified by their public keys. An experimenter can perform actions on the sandbox such as running experiments, reassigning the sandbox's resources to another sandbox, or allowing other experimenters to access the sandbox.

4.1.3 Device manager

The Tsumiki software that runs on an device owner's system consists of the sandbox, the resource manager, and the device manager. The device manager keeps components up-to-date to improve security and robustness without any action by the device owner. The device manager is written to only allow properly signed updates provided by our research group (the core software maintainers). It is also written to be safe from various known attacks on software update systems [11]. Currently, we have three device manager implementations, for Seattle on laptops and desktops, for Seattle on Android, and for Sensibility Testbed on Android, respectively.

4.1.4 Experiment manager

Seash. An experiment manager provides experimenters with a simple interface for interacting with the resource manager on remote devices. The experiment manager is used by an experimenter to easily control and monitor the sandboxes in a testbed. The primary interactive service manager used in Tsumiki is `seash` [58]. It provides an interactive shell to the experimenters.

Overlord. It is difficult to maintain a long-running experiment as the devices hosting it may not be available at all times. To deploy an arbitrary experiment on a number of devices and ensure the experiment is up and running, we developed Overlord [41], a non-interactive experiment manager. It can be used to script the interaction between the clearinghouses and devices to keep a long-lasting experiment online. Overlord is used to deploy many of the infrastructure services for Tsumiki.

HuXiang. The HuXiang experiment manager [1] runs on remote devices and register a common domain name with a dynamic DNS service, or announce the same key using another lookup service. When their browser resolves

[6]We broadly define sensors as the hardware components that can record phenomena about the physical world, such as the WiFi/cellular network, GPS location, movement acceleration, etc.

the domain name or looks up the key, users are directed to one of the server instances, with the total load distributed over all participating nodes. Since nodes may join and leave HuXiang over time, the web content can also evade IP address filters.

Try Repy! This is a web-based Integrated Development Environment (IDE) for Tsumiki's Repy sandbox deployment on remote devices. It provides a visual editor with auto-indent, line numbering, and different source code syntax highlighting styles inside a browser tab. An experimenter can use Try Repy! to quickly type in experiment code on a web interface, and use Try Repy!'s GUI to run experiments and view logs.

Seastorm. This experiment manager with visualization capabilities [4] was initially created as a tool to support teaching of distributed algorithms. It displays the execution of algorithms as interactive sequence diagrams, thereby making reasoning about and debugging these algorithms easier [35]. Seastorm runs in the browser on any platform, and requires no manual modification of the experimenter's algorithms in order to support visualization.

Owl. Owl is a service monitor which can be used to monitor complex, long running experiments. This tool allows an experimenter to send application specific data to a central Owl server, so that an experimenters may gain better insight into his running experiments, such as device status, experiment logs, etc. Owl was originally developed to monitor experiments on the PlanetLab but has since been ported for use with Tsumiki [42].

4.1.5 Installer builder

Any experimenter can easily obtain a customized installer for Tsumiki which provides access to sandboxes in any way the experimenter specifies. These can be given out by an experimenter who does not want use a clearinghouse. In addition, these installers can be bundled with applications to allow the experimenter direct control over safe experimental sandboxes on their end user devices.

4.1.6 Clearinghouse

Seattle clearinghouse. The common model for end users participation in Seattle is to associate a public key with the controlling experimenter of a sandbox corresponding to a clearinghouse. This allows the clearinghouse to have ultimate control over which experimenters are allowed to access a sandbox. Our clearinghouse provides a website, through which experimenter can request sandbox on remote devices [23]. When our clearinghouse discovers a new Tsumiki installation, it determines which experimenter account provided the sandbox. That experimenter is credited with the ability to obtain ten additional sandboxes on our testbed as long as that installation is still functional. In our experience this incentive has worked especially well for bootstrapping an experimenter with sufficient sandboxes to carry out interesting experiments.

Social clearinghouse and **Sensibility testbed clearinghouse** have been introduced in Section 7 and [19], respectively.

4.1.7 Lookup service

A lookup service allows clearinghouses and experiment manager to locate the corresponding sandboxes. We currently use several lookup services to provide a level of robustness to the system [16].

OpenDHT. We currently use OpenDHT [53] as a distributed way to store data. Each sandbox advertise their availability via OpenDHT using a public key. OpenDHT runs on a variety of PlanetLab nodes and replicates data between them to handle failures.

DOR. The Digital Object Registry (DOR) [34] is a centralized service that stores digital objects in a distributed network. Each digital object is represented by a unique identifier that could contain a cryptographic hash or fingerprint of the identified object. DOR can also be used as a key-value store, which is the way that we use it in Tsumiki as a lookup service.

Centralized lookup service. We implemented a central lookup service which is a centralized hash table abstraction. This is used to aid in lookup redundancy in case either OpenDHT or DOR fails. We currently have three implementations of the central lookup service: using UDP, Repy V1 and Repy V2, respectively.

Secure lookup service. A student at the University of Washington is currently building a new lookup service that will provide integrity guarantees for data. Users of the lookup service will only be able to put values under a location in the key-value store if they have the corresponding private keys. This will allow users to easily obtain a portion of the key space that is unique to them. This allows the authentication of services on our testbed.

Note that in addition to the human-visible interfaces varying wildly, the roles and functionality of a component may also vary wildly. For example, the overlord experiment manager [41] executes code that the experimenter provides and uses this to decide how to deploy service instances. So while a task like executing code is something one might expect only a sandbox would do, human-visible interfaces can provide enough flexibility to support diverse component realizations.

4.2 Auxiliary services

In addition to the core components, there are other tools available to the system and to its users that serve auxiliary purposes.

Affix networking library. The established programming abstraction for networks is the venerable Berkeley socket API, which does not support mobility, NAT traversal, and so on. An Affix component encapsulates diverse network functionalities in a library that obeys the network API's semantics. An Affix component or stack of Affix components can be used in place of the current network API, without requiring the application to be modified.

Time service. Various aspects of a testbed benefit from loosely synchronized clocks on communicating systems, including coordinating measurement times and verifying the timeliness of cryptographic signatures. To support devices blocked from sending NTP traffic and decrease the reliance on communication outside of the testbed, a time service is built. This service provides functionality similar to NTP without requiring the use of public NTP servers.

Zenodotus. To run a service in Tsumiki it is convenient to have a network name rather than a set of (possible changing) IP addresses to access a service. Using a dynamic DNS server Zenodotus, an experiment can have a stable URL despite that devices are mobile. Zenodotus achieves this by reading network names out of the lookup service to answer DNS queries. Zenodotus also supports delegation queries to another name server of the experimenter's choosing. This can be used to build other services like load balancers or geo-location services.

GeoIP geo-location service. Modern services often detect the geographic region a device is located in to offer localized content. With the GeoIP service, location-based services can be constructed on Tsumiki. GeoIP is also used in the experiment manager `seash` to give experimenters an indication where the devices they have access to are located.

5 Experiences with existing testbeds

Our design of Tsumiki components is drawn from experiences with existing testbeds and their functionality reuse and extension. 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, as shown in Table 6. Note that this table is our interpretation of the existing testbed construction, by performing the analysis based on the cited papers. Therefore, there may be inaccuracies as we are not intimately familiar with all design aspects of each system.

From the table, it is clear that most of the existing testbeds can be (at least partially) mapped into the Tsumiki model. Therefore, the Tsumiki model subsumes all of these prior testbeds in that its components can well represent the current testbeds. Using both the experience of prior testbeds and our own, Tsumiki's components can provide testbed developers the flexibility to reuse existing functionality, and Tsumiki's interfaces can allow testbed developers to cater the components to the requirements of their research.

6 Example 1: Replicating mobile experiments

To investigate whether testbeds constructed with Tsumiki can operate correctly and be used for non-trivial experiments, we replicated the sensor-augmented WiFi protocol published in NSDI'11 [50] and evaluated it on the Sensibility Testbed [19]. The following is a brief overview of three wireless algorithms introduced in [19], and an explanation of how we adapted them to run on the Sensibility testbed. All three algorithms are implemented at the WiFi sender.

6.1 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 (30ms in our implementation), the sender samples a higher rate and switches to it if it is the highest rate and has not failed in the recent past (50ms in our case).

Testbed	Sandbox	Resource Manager	Device Manager	Experiment manager	Installer builder	Clearinghouse	Lookup service	Reference
BISmark	BISmark routers*	ssh server on BISmark router	Custom package management utilities on top of OpenWrt's built-in opkg	Data collection servers, `bismark-data-transmit`	Package repository servers	User registration system (in firmware)	BISmark router sends heartbeats to monitoring servers	[62, 63]
BOINC	BOINC client	Python scripts, C++ interfaces, general preferences	BOINC installer tool	Scheduling servers, Data servers, Client GUI	Anonymous platform mechanism	Project master URL	N/A	[7]
Dasu	Dasu client's experiment sandbox	Configuration service	BitTorrent extension	Data service, Secondary experiment administration service	N/A	Primary experiment administration service	Coordination service	[57]
DETER	Computing elements, Container	Virtualization engine	Embedder	SEER (workbench), Experiment lifecycle manager (ELM)	N/A			[12, 39]
DIMES	DIMES agent	Agent's scheduling mechanism	Auto-update mechanism	Experiment planning system	N/A	Management system	Discovery	[60]
Emulab	Virtual nodes, OpenVZ linux containers, Xen†	Global resource allocation	Node configuration	Emulab web interface, `ns`	`masterhost`	Authorization structure	N/A	[69]
Fathom	JavaScript extension for Firefox, Fathom object and APIs	Worker thread	Firefox add-ons manager	Firefox browser tab	Packaged Fathom script	N/A	N/A	[26]
Flexlab	Emulab container, Network model	Application monitor	Same as Emulab	Emulab portal, Experimentation workbench‡	`masterhost`, Measurement repository	Same as Emulab	N/A	[54]
ModelNet	Virtual edge node, core node	Assignment, Binding phases	Run phase	N/A	Create, Distillation phases	N/A	N/A	[67]
PlanetLab	VM, slice	Node manager	Node manager, boot CD	Slice creation service, `ssh`	PlanetLab Central (PLC)			[43, 44]
RON	RON client, `conduit` API△	`forwarder` object	N/A	N/A	RON router (implements routing protocol)	N/A	RON membership manager, flooder	[6]
SatelliteLab Planet	Same as PlanetLab							[27]
Satellite	JVM (does not permit code execution)	N/A	N/A	Manual	OS-specific installers	Manual allocation	Heartbeat mechanism°	
SPLAY	Sandboxed◇ process forked by `splayd` and libraries	Daemon `splayd`, churn management	N/A	Command line, Web interface	`jobs` process	Controller `splayctl`	`ctl` process (keeps track of all daemons and processes)	[36]
VINI	PlanetLab VServers, extended CPU scheduling, virtual network devices	Same as PlanetLab (with extensions such as virtual point-to-point connectivity, distinct forwarding tables and routing processes per virtual node, etc.)						[10]

*Netgear WNDR 3800 routers running a custom version of OpenWrt distribution.

†`https://wiki.emulab.net/wiki/vnodes`.

‡"An Experimentation Workbench for Replayable Networking Research", NSDI'07 [28].

△The API that a RON client uses to send and receive packets.

°The satellite helper periodically sends a small status message to its planet proxy.

◇A Lua-based sandbox.

Table 6: 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.

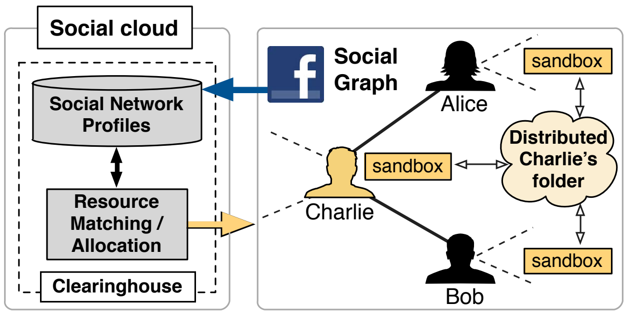


Figure 6: A peer-to-peer storage use case of the Social Compute Cloud. Charlie is friends with Alice and Bob on Facebook. The testbed provides Charlie with a distributed folder that is replicated to devices controlled by Alice and Bob.

6.2 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 have been periodically sampled. Over the past 10s, if the sampled rate exhibits better performance (lower packet loss and lower transmission time) than the current rate, SampleRate switches to this rate.

6.3 Hint-aware adaptation algorithm (hybrid)

This algorithm uses RapidSample for data transmission when the receiver is mobile, but uses SampleRate when the receiver is stationary. It relies on movement hints generated by the movement detection algorithm that are piggybacked on acknowledgement packets to the sender and are interpreted at the sender as a signal to switch between the mobile and stationary algorithms. At the receiver, we use a smartphone accelerometer to detect movement.

7 Example 2: Social compute cloud

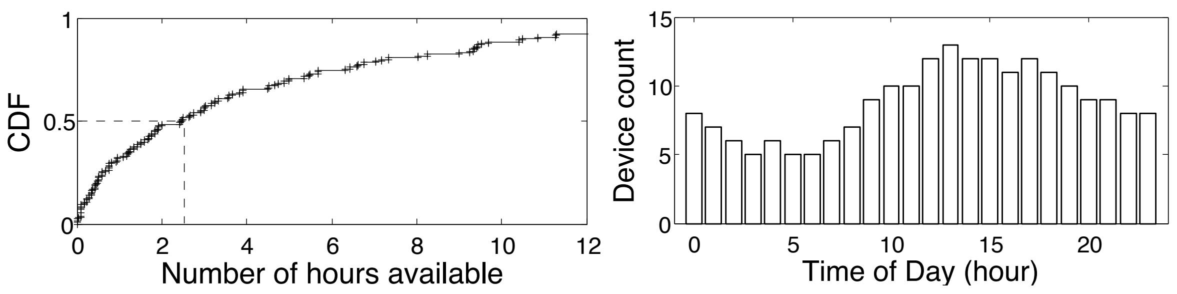
Inspired by the trust associated with interpersonal relationships formed within a social network, a group of researchers from the University of Chicago and Karlsruhe Institute of Technology developed the Social Compute Cloud [20–22]. This system uses social network platforms to locate computational and storage resources within one's social circle, and uses social preference matching algorithms to pair resource providers with consumers. Figure 6 illustrates a use case in which the Social Compute Cloud provides a friend-to-friend storage service (similar to PAST [55] or CFS [25]). In this P2P system, the burden of hosting data is transferred to peers' devices within one's social network.

In this section we detail how Tsumiki components were used to build the Social Compute Cloud testbed and then use this testbed to measure the availability of resources within an individual's social network.

Reused or customized Tsumiki components. The Social Compute Cloud builds on several Tsumiki components. It uses the *sandbox*, *resource manager*, *device manager*, and *installer builder* to provide sandboxing, virtualization, and software packaging, respectively. Similarly, it uses the *lookup service* to manage and discover testbed members and their resource contributions, and the *experiment manager* to access remote resources.

This testbed uses a custom *clearinghouse* to integrate with social networks and to provide socially-aware resource allocations. This includes Facebook-based authentication, association between social identities and Tsumiki identities, and using Facebook access tokens to access a users' social network and their sharing preferences. The researchers also developed several new human-visible interfaces to enable users to define sharing preferences based on their social relationship. The customized clearinghouse uses these preferences and an external preference matching service to determine mappings among resource consumers and providers. Following resource allocation, standard Tsumiki mechanisms are used to manage resources.

Experiences with integration. The modifications took about one month. The majority of the development effort focused on implementing the social clearinghouse. This involved 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.



(a) CDF of availability of friend devices. (b) Number of available friend devices at different hours.

Figure 7: Aggregate friend device availability across all days in the study as (a) a CDF, and (b) median count of available devices for each hour in a day.

Resources in a social network. In this section, we evaluate the Social Compute Cloud testbed via the deployment of a friend-to-friend backup service. This service provides storage and computation resources from one's social network. To design a robust version of a friend-to-friend backup service, a researcher must have a detailed understanding of friends' device availability. This information can influence replication policy, device selection, and other design factors. In this section we describe the results from a 5-day experiment (Sun — Thurs) deployed on the Social Compute Cloud testbed.

To conduct the experiment, recruited participants were asked to install Social Compute Cloud on their personal devices. This resulted in 17 device installations in total. These included laptops, desktops, and mobile phones. We then deployed an experiment in which each device sent a ping every 5 minutes to a server located at the university. We then used the server's ping records to determine 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 circle on Facebook.

Figure 7(a) shows an aggregate view of the availability data across all devices as a CDF (five desktops, which had uninterrupted availability, are not shown). This figure indicates that the median continuous availability period across the devices was at least 2.5 hours. A friend-to-friend backup service can use this information for selective file placement (e.g., to parameterize a policy that places the most recently touched files on devices with the highest availability). Figure 7(b) considers the device availability data by hour, aggregated across all five days (due to the small number of days, we do not include standard deviation). This figure shows that for each hour between 10 AM and 7 PM the median number of available devices was at least 10. In a friend-to-friend backup service this hour-by-hour availability data can be used for scheduling data replication between devices. In the experiment, the hours 10AM–7PM are especially relevant because during these hours many of the devices are co-located on the university campus. During this time, inter-device latency is low and inter-device bandwidth is high, making this period especially suitable for inter-device replication.

Both graphs in Figure 7 capture device activity due to user mobility. For example, Figure 7(b) shows that users connect and disconnect at predictable times in the day. These results demonstrate that the Social Compute Cloud testbed is particularly useful for deployments that rely on data about social network users' activity, such as their mobility and connectivity patterns.

8 Example 3: Seattle

The Seattle testbed [59] was released in 2009 and uses all of the default Tsumiki components. The Seattle clearinghouse implements a policy that allows an experimenter to use 10 sandboxes at a time. Experimenters are allowed to use 10 additional sandboxes for every active installation that is linked to their account. This provides an incentive to fuel adoption and growth of the platform. Researchers and educators interested in using Seattle without contributing resources to the global resource pool have used the Tsumiki component model to run private testbed instances.

Cumulated over its operational history so far, Seattle has served updates to about 39K geographically distributed

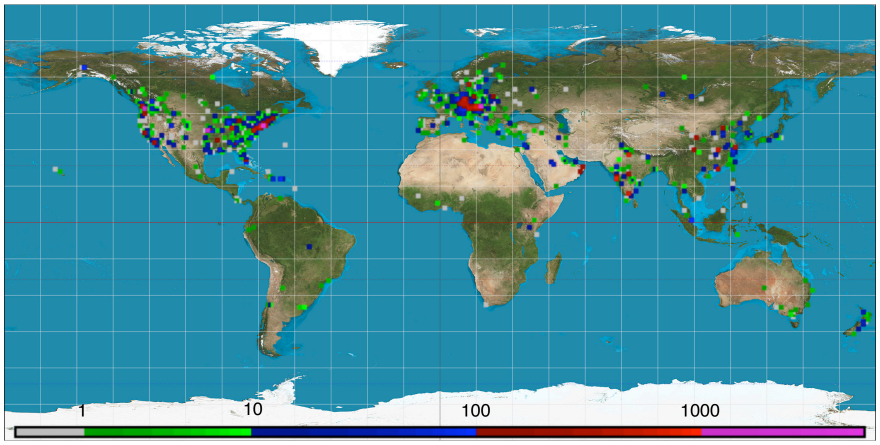


Figure 8: Seattle node location derived by applying GeoIP to the addresses that contact the software updater.

devices, including smartphones, tablets, and laptops[7]. Figure 8 shows that these devices are distributed world-wide, with China, India, Austria, Germany, and the US each with over 1K nodes. We ran reverse DNS lookups on Seattle nodes' IP addresses to categorize them at a high level. Most devices either do not respond to reverse DNS lookups or have names that explicitly indicate a home node (77%). A substantial number of devices (5,847) are located on university networks (14%), and 831 devices are associated with other testbeds (2%). Of the total number, 6,829 devices (17%) report platform names indicative of mobile devices, such as smartphones and tablets.

This device diversity and open access has allowed Seattle to provide a diverse platform for research on end user devices [24, 29, 30, 38, 40, 65, 66, 68, 71]. In addition, Seattle has been used in over 60 classes, including classes on cloud computing, operating systems, security, and networking [13, 14, 18, 32, 68]. Over its seven years of operation, more than four thousand researchers have used the Seattle clearinghouse.

9 Example 4: BISmark

Home networks are difficult to measure and understand. This is due to many factors. For example, many home devices are typically hidden behind a NAT, there exists wireless interference between devices, and devices frequently join and leave the network. The BISmark project seeks to measure and understand home networks by providing users with wireless routers that let researchers measure users' home network [62, 63]. This deployment faces many challenges, including convincing users to deploy BISmark routers in their homes and keeping BISmark routers online. Since the BISmark router is on the path of the user's communication, a poorly controlled experiment can cripple a user's home Internet connection.

We worked with the BISmark team to do a proof-of-concept version of BISmark using Tsumiki components. The resulting testbed uses the unchanged versions of the *experiment manager* and *lookup services*. BISmark also adopts the existing *resource manager*, *device manager*, and *sandbox* 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) to fit within the storage on the platform. The component interfaces did not change in any way.

Experiences with integration. The needed changes were mostly performed over a two day period when the first author visited the BISmark team. The changes were primarily in three areas. First, it was necessary to patch several minor portability problems with Tsumiki components due to differences in the OpenWrt environment (the core BISmark software is developed on), and other supported OSes (namely Linux). Second, it was necessary to reduce the amount of storage space needed because the devices have only 16MB of flash storage. This space is not even enough to install OpenWrt and a normal Python install. It necessitated building a device manager which reduced the on-device footprint and setting up a separate installer builder to update the space-optimized version of the software components that runs on the device. Third, the sandbox needed to add support for additional functionality, such as `traceroute` and

[7]This number over-estimates devices that change their IP, while under-estimating devices behind NATs and devices that utilize a different install builder.

packet capture. Adding functionality available via external shell commands or libraries is relatively straightforward. We chose to add `traceroute` to validate that adding sandbox functionality did not add any additional complexity on this platform. All other components operate and interoperate smoothly because the interfaces between components did not change.

By using Tsumiki, BISmark team can use any of the experiment managers to quickly and easily deploy their code. If one of their routers is moved to a different network, the lookup service will automatically be notified. In particular, by running experiments in a Tsumiki sandbox, BISmark experiments are security and performance isolated. The use of Tsumiki components will make it easier to support BISmark's goals of non-interference with a user's home network.

10 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.

References

[1] Huxiang. Accessed February 2, 2016, `https://seattle.poly.edu/wiki/huxiang`.

[2] Node manager design document. Accessed February 2, 2016, `https://seattle.poly.edu/wiki/UnderstandingSeattle/NodeManagerDesign`.

[3] Openwrt – wireless freedom. Accessed February 2, 2016, `https://openwrt.org/`.

[4] Seastorm source code repository. Accessed February 2, 2016, `https://github.com/jakobkallin/Seattle-Seastorm`.

[5] Seattle clearinghouse. Accessed February 2, 2016, `https://seattleclearinghouse.poly.edu/`.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. of SOSP '01*, Banff, Alberta, Canada.

[7] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.

[8] S. Banerjee, T. G. Griffin, and M. Pias. The interdomain connectivity of PlanetLab nodes. In *Proceedings of the 5th Passive and Active Measurement Conference (PAM)*, Apr 2004.

[9] C. Barsan and J. Cappos. Containing Node Communication In Seattle. `https://seattle.poly.edu/wiki/ContainmentInSeattle`.

[10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 3–14. ACM, 2006.

[11] A. Bellissimo, J. Burgess, and K. Fu. Secure Software Updates: Disappointments and New Challenges. In *1st USENIX Workshop on Hot Topics in Security*, pages 37–43, Vancouver, Canada, 2006.

[12] T. Benzel. The science of cyber security experimentation: the deter project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 137–148. ACM, 2011.

[13] J. Cappos and I. Beschastnikh. Teaching networking and distributed systems with seattle: tutorial presentation. *J. Comput. Sci. Coll.*, 25(5):308–310, May 2010.

[14] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Seattle: A platform for educational cloud computing. In *The 40th Technical Symposium of the ACM Special Interest Group for Computer Science Education (SIGCSE '09)*, 2009.

[15] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *The 17th ACM Conference on Computer and Communications Security (CCS '10)*, 2010.

[16] J. Cappos and J. Hartman. Why It Is Hard to Build a Long Running Service on Planetlab. In *WORLDS*, 2005.

[17] J. Cappos, L. Wang, R. Weiss, Y. Yang, and Y. Zhuang. Blursense: Dynamic fine-grained access control for smartphone privacy. In *Sensors Applications Symposium (SAS)*. IEEE, 2014.

[18] J. Cappos and R. Weiss. Teaching the security mindset with reference monitors. *SIGCSE Bull.*, 45(1), 2014.

[19] J. Cappos, Y. Zhuang, A. Rafetseder, and I. Beschastnikh. Tsumiki: A Meta-Platform for Building Your Own Testbed. 2015. submitted to USENIX ATC'16.

[20] S. Caton, C. Haas, K. Chard, K. Bubendorfer, and O. Rana. A social compute cloud: Allocating and sharing infrastructure resources via social networks. In *IEEE Transactions on Services Computing*. IEEE, 2014.

[21] K. Chard, K. Bubendorfer, S. Caton, and O. Rana. Social cloud computing: A vision for socially motivated resource sharing. *Services Computing, IEEE Transactions on*, 5(4):551–563, Fourth 2012.

[22] K. Chard, S. Caton, O. Rana, and K. Bubendorfer. Social cloud: Cloud computing in social networks. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 99–106. IEEE, 2010.

[23] Seattle Clearinghouse. `https://seattleclearinghouse.poly.edu/`.

[24] L. Collares, C. Matthews, J. Cappos, Y. Coady, and R. McGeer. Et (smart) phone home! In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11*, pages 283–288. ACM, 2011.

[25] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. *SIGOPS Oper. Syst. Rev.*, 35(5):202–215, Oct. 2001.

[26] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: A browser-based network measurement platform. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 73–86. ACM, 2012.

[27] M. Dischinger, A. Haeberlen, I. Beschastnikh, K. P. Gummadi, and S. Saroiu. Satellitelab: adding heterogeneity to planetary-scale network testbeds. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 315–326. ACM, 2008.

[28] E. Eide, L. Stoller, and J. Lepreau. An experimentation workbench for replayable networking research. In *NSDI*, 2007.

[29] J. Eisl, A. Rafetseder, and K. Tutschku. Service architectures for the future converged internet: Specific challenges and possible solutions for mobile broad-band traffic management. *Advances in Next Generation Services and Service Architectures*, Oct. 2010.

[30] P. M. Eittenberger, M. Großmann, and U. R. Krieger. Doubtless in seattle: Exploring the internet delay space. In *Next Generation Internet (NGI), 2012 8th EURO-NGI Conference on*, pages 149–155. IEEE, 2012.

[31] C. Elliott. Geni-global environment for network innovations. In *LCN*, page 8, 2008.

[32] S. Hooshangi, R. Weiss, and J. Cappos. Can the security mindset make students better testers? In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 404–409. ACM, 2015.

[33] X. Huang. An experimental platform for home wifi routers. Master's Thesis, NYU, 2016. In preparation.

[34] R. E. Kahn and P. A. Lyons. Representing value digital objects: A discussion of transferability and anonymity. *J. on Telecomm. & High Tech. L.*, 5:189, 2006.

[35] J. Kallin. Visualizing Distributed Algorithms on the Seattle Platform. Master's Thesis at the University of Gothenburg, Sweden. 2014. Supervisor, Examiner: Olaf Landsiedel.

[36] L. Leonini, É. Rivière, and P. Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI*, volume 9, pages 185–198, 2009.

[37] T. Li, A. Rafetseder, R. Fonseca, and J. Cappos. Fence: protecting device availability with uniform resource control. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, pages 177–191. USENIX Association, 2015.

[38] F. Metzger, A. Rafetseder, D. Stezenbach, and K. Tutschku. Analysis of web-based video delivery. In *FITCE Congress (FITCE), 2011 50th*, pages 1–6. IEEE, 2011.

[39] J. Mirkovic, T. V. Benzel, T. Faber, R. Braden, J. T. Wroclawski, and S. Schwab. The deter project: Advancing the science of cyber security experimentation and test. In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 1–7. IEEE, 2010.

[40] P. Müller, D. Schwerdel, and J. Cappos. Tomato a virtual research environment for large scale distributed systems research. *PIK-Praxis der Informationsverarbeitung und Kommunikation*, pages 1–10.

[41] Overlord Deployment and Monitoring Library. Accessed February 2, 2016, `https://seattle.poly.edu/wiki/Libraries/Overlord`.

[42] Owl support for seattle. `https://seattle.poly.edu/blog/Owl%20running%20on%20Seattle!`

[43] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *HotNets*, 2002.

[44] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366. USENIX Association, 2006.

[45] H. Pucha, Y. C. Hu, and Z. M. Mao. On the imapact of research network based testbeds on wide-area experiments. In *Proceedings of IMC'06*, Oct 2006.

[46] L. Pühringer. *Try Repy!* PhD thesis, Bachelors thesis, University of Vienna, 2011.

[47] All of the python you need to forget to use repy. `https://seattle.poly.edu/wiki/PythonVsRepy`.

[48] A. Rafetseder, F. Metzger, L. Pühringer, K. Tutschku, Y. Zhuang, and J. Cappos. Sensorium–a generic sensor framework. *Praxis der Informationsverarbeitung und Kommunikation*, 36(1):46–46, 2013.

[49] J. Rasley, E. Gessiou, T. Ohmann, Y. Brun, S. Krishnamurthi, and J. Cappos. Detecting latent cross-platform api violations. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE, 2015.

[50] L. Ravindranath, C. Newport, H. Balakrishnan, and S. Madden. Improving wireless network performance using sensor hints. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, 2011.

[51] Repy Programming Guide. `https://seattle.poly.edu/wiki/RepyApi`.

[52] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. Opendht: A public dht service and its uses. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, New York, NY, USA, 2005. ACM.

[53] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. *ACM SIGCOMM Computer Communication Review*, 35(4):73–84, 2005.

[54] R. Ricci, J. Duerig, P. Sanaga, D. Gebhardt, M. Hibler, K. Atkinson, J. Zhang, S. K. Kasera, and J. Lepreau. The Flexlab Approach to Realistic Evaluation of Networked Systems. In *NSDI*, 2007.

[55] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *SIGOPS Oper. Syst. Rev.*, 35(5):188–201, Oct. 2001.

[56] SamKnows. `https://www.samknows.com/`.

[57] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing experiments to the internet's edge. In *USENIX NSDI*, volume 2013, pages 487–499, 2013.

[58] Seash: The Seattle Shell. `https://seattle.poly.edu/wiki/SeattleShell`.

[59] Seattle homepage. Accessed February 2, 2016, `https://seattle.poly.edu`.

[60] Y. Shavitt and E. Shir. Dimes: Let the internet measure itself. *ACM SIGCOMM Computer Communication Review*, 35(5):71–74, 2005.

[61] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using planetlab for network research: myths, realities, and best practices. *SIGOPS Oper. Syst. Rev.*, 40(1):17–24, 2006.

[62] S. Sundaresan, S. Burnett, N. Feamster, and W. De Donato. BISmark: A testbed for deploying measurements and applications in broadband access networks. In *USENIX Annual Technical Conference*, 2014.

[63] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband internet performance: a view from the gateway. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 134–145. ACM, 2011.

[64] ToMaTo Testbed. `http://tomato-lab.org/`.

[65] S. Tredger, Y. Zhuang, C. M. J. Short-Gershman, Y. Coady, and R. McGeer. Building green systems with green students: An educational experiment with geni infrastructure. In *Research and Educational Experiment Workshop (GREE), 2013 Second GENI*, pages 29–36. IEEE, 2013.

[66] K. Tutschku, A. Rafetseder, J. Eisl, and W. Wiedermann. Towards sustained multi media experience in the future mobile internet. In *2010 14th International Conference on Intelligence in Next Generation Networks (ICIN)*, Berlin, Germany, Oct. 2010.

[67] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[68] S. A. Wallace, M. Muhammad, J. Mache, and J. Cappos. Hands-on internet with seattle and computers from across the globe. *J. Comput. Sci. Coll.*, 27(1):137–142, Oct. 2011.

[69] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[70] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79 –93, may 2009.

[71] Y. Zhuang, C. Matthews, S. Tredger, S. Ness, J. Short-Gershman, L. Ji, N. Rebenich, A. French, J. Erickson, K. Clarkson, et al. Taking a walk on the wild side: teaching cloud computing on distributed research testbeds. In *Proceedings of the 45th ACM technical symposium on Computer science education*, pages 535–540. ACM, 2014.

[72] Y. Zhuang, A. Rafetseder, L. Wang, I. Beschastnikh, and J. Cappos. Sensibility testbed: an internet-wide cloud platform for programmable exploration of mobile devices. In *INFOCOM 2014, Poster*.