SPLAY: Distributed Systems Evaluation Made Simple[*]

(or how to turn ideas into live systems in a breeze)

Lorenzo Leonini[†] Étienne Rivière[‡] Pascal Felber
University of Neuchâtel, Switzerland

Abstract

This paper presents SPLAY*, an integrated system that facilitates the design, deployment and testing of large-scale distributed applications. Unlike existing systems,* SPLAY *covers all aspects of the development and evaluation chain. It allows developers to express algorithms in a concise, simple language that highly resembles pseudo-code found in research papers. The execution environment has low overheads and footprint, and provides a comprehensive set of libraries for common distributed systems operations.* SPLAY *applications are run by a set of daemons distributed on one or several testbeds. They execute in a sandboxed environment that shields the host system and enables* SPLAY *to also be used on non-dedicated platforms, in addition to classical testbeds like PlanetLab or ModelNet. A controller manages applications, offering multi-criterion resource selection, deployment control, and churn management by reproducing the system's dynamics from traces or synthetic descriptions.* SPLAY*'s features, usefulness, performance and scalability are evaluated using deployment of representative experiments on PlanetLab and ModelNet clusters.*

1 Introduction

Developing large-scale distributed applications is a highly complex, time-consuming and error-prone task. One of the main difficulties stems from the lack of appropriate tool sets for quickly prototyping, deploying and evaluating algorithms in real settings, when facing unpredictable communication and failure patterns. Nonetheless, evaluation of distributed systems over real testbeds is highly desirable, as it is quite common to discover discrepancies between the expected behavior of an application as modeled or simulated and its actual behavior when deployed in a live network.

While there exist a number of experimental testbeds to address this demand (e.g., PlanetLab [11], ModelNet [35], or Emulab [38]), they are unfortunately not used as systematically as they should. Indeed, our first-hand experience has convinced us that it is far from straightforward to develop, deploy, execute and monitor applications for them and the learning curve is usually slow. Technical difficulties are even higher when one wants to deploy an application on several testbeds, as deployment scripts written for one testbed may not be directly usable for another, e.g., between PlanetLab and ModelNet. As a side effect of these difficulties, the performance of an application can be greatly impacted by the technical quality of its implementation and the skills of the person who deploys it, overshadowing features of the underlying algorithms and making comparisons potentially unsound or irrelevant. More dramatically, the complexity of using existing testbeds discourages researchers, teachers, or more generally systems practitioners from fully exploiting these technologies.

These various factors outline the need for novel development-deployment systems that would straightforwardly exploit existing testbeds and bridge the gap between algorithmic specifications and live systems. For researchers, such a system would significantly shorten the delay experienced when moving from simulation to evaluation of large-scale distributed systems ("time-to-paper" gap). Teachers would use it to focus their lab work on the core of distributed programming—algorithms and protocols—and let students experience distributed systems implementation in real settings with little effort. Practitioners could easily validate their applications in the most adverse conditions.

There already exist several systems to ease the development or deployment process of distributed applications. Tools like Mace [23] or P2 [26] assist the developer by generating code from a high-level description, but do not provide any facility for its deployment or evaluation. Tools such as Plush [9] or Weevil [37] help for the deployment process, but are restricted to situations where the user has control over the nodes composing the testbed (i.e., the ability to run programs remotely using `ssh` or similar).

To address these limitations, we propose SPLAY, an infrastructure that simplifies the prototyping, development, deployment and evaluation of large-scale systems. Unlike existing tools, SPLAY covers the whole chain of distributed systems design and evaluation. It allows developers to specify distributed applications in a concise manner using a platform-independent, lightweight and efficient language based on Lua [20]. For instance, a complete implementation of the Chord [33] distributed hash table (DHT) requires approximately 100 lines of code.

SPLAY provides a secure and safe environment for executing and monitoring applications, and allows for a simplified and unified usage of testbeds such as PlanetLab, ModelNet, networks of idle workstations, or per-

[*]This work is supported in part by the Swiss National Foundation under agreement number 102819.

[†]Contact author: `lorenzo.leonini@unine.ch`

[‡]This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

sonal computers. SPLAY applications execute in a safe, sandboxed environment with controlled access to local resources (file system, network, memory) and can be instantiated on a large set of nodes with a single command. SPLAY supports multi-user resource reservation and selection, orchestrates the deployment and monitors the whole system. It is particularly easy with SPLAY to reproduce a given live experiment or to control several experiments at the same time.

An important component of SPLAY is its churn manager, which can reproduce the dynamics of a distributed system based on real traces or synthetic descriptions. This aspect is of paramount importance, as natural churn present in some testbeds such as PlanetLab is not reproducible, hence preventing a fair comparison of protocols under the very same conditions.

SPLAY is designed for a broad range of usages, including: (i) deploying distributed systems whose lifetime is specified at runtime and usually short, e.g., distributing a large file using BitTorrent [17]; (ii) executing long-running applications, such as an indexing service based on a DHT or a cooperative web cache, for which the population of nodes may dynamically evolve during the lifetime of the system (and where failed nodes must be replaced automatically); or (iii) experimenting with distributed algorithms, e.g., in the context of hands-on networking class, by leveraging the isolation properties of SPLAY to enable execution of (possibly buggy) code on a shared testbed without interference.

Contributions. This paper introduces a distributed infrastructure that greatly simplifies the prototyping, development, deployment, and execution of large-scale distributed systems and applications. SPLAY includes several original features—notably churn management, support for mixed deployments, and platform-independent language and libraries—that make the evaluation and comparison of distributed systems much easier and fairer than with existing tools.

We show how SPLAY applications can be concisely expressed with a specialized language that closely resembles the pseudo-code usually found in research papers. We have implemented several well-known systems: Chord [33], Pastry [31], Scribe [15], SplitStream [14], BitTorrent [17], Cyclon [36], Erdös-Renyi epidemic broadcast [19] and various types of distribution trees [13].

Our system has been thoroughly evaluated along all its aspects: conciseness and ease of development, efficiency, scalability, stability and features. Experiments convey SPLAY's good properties and the ability of the system to help practitioner and researcher alike through the whole distributed system design, implementation and evaluation chain.

Roadmap. The remaining of this paper is organized as follows. We first discuss related work in Section 2. Section 3 gives an overview of the SPLAY architecture and elaborates on its design choices and rationales. In Section 4, we illustrate the development process of a complete application (the Chord DHT [33]). Section 5 presents a complete evaluation of SPLAY, using representative experiments and deployments (including tests of the Chord implementation of Section 4). Finally, we conclude in Section 6.

2 Related Work

SPLAY shares similarities with a large body of work in the area of concurrent and distributed systems. We only present systems that are closely related to our approach.

Development tools. On the one hand, a set of new languages and libraries have been proposed to ease and speed up the development process of distributed applications.

Mace [23] is a toolkit that provides a wide set of tools and libraries to develop distributed applications using an event-driven approach. Mace defines a grammar to specify finite state machines, which are then compiled to C++ code, implementing the event loop, timers, state transitions, and message handling. The generated code is platform-dependent: this can prove to be a constraint in heterogeneous environments. Mace focuses on application development and provides good performance results but it does not provide any built-in facility for deploying or observing the generated distributed application.

P2 [26] uses a declarative logic language named OverLog to express overlays in a compact form by specifying data flows between nodes, using logical rules. While the resulting overlay descriptions are very succinct, specifications in P2 are not natural to most network programmers (programs are largely composed of table declaration statements and rules) and produce applications that are not very efficient. Similarly to Mace, P2 does not provide any support for deploying or monitoring applications: the user has to write his/her own scripts and tools.

Other domain-specific languages have been proposed for distributed systems development. In RTAG [10], protocols are specified as a context-free grammar. Incoming messages trigger reduction of the rules, which express the sequence of events allowed by the protocol. Morpheus [8] and Prolac [24] target network protocols development. All these systems share the goal of SPLAY to provide easily readable yet efficient implementations, but are restricted to developing low-level network protocols, while SPLAY targets a broader range of distributed systems.

Deployment tools. On the other hand, several tools have been proposed to provide runtime facilities for distributed applications developers by easing the deployment and monitoring phase.

Neko [34] is a set of libraries that abstract the network substrate for Java programs. A program that uses Neko can be executed without modifications either in simulations or in a real network, similarly to the NEST testbed [18]. Neko addresses simple deployment issues,

by using daemons on distant nodes to launch the virtual machines (JVMs). Nonetheless, Neko's network library has been designed for simplicity rather than efficiency (as a result of using Java's RMI), provides no isolation of deployed programs, and does not have built-in support for monitoring. This restricts its usage to controlled settings and small-scale experiments.

Plush [9] is a set of tools for automatic deployment and monitoring of applications on large-scale testbeds such as PlanetLab [11]. Applications can be remotely compiled from source code on the target nodes. Similarly to Neko and SPLAY, Plush uses a set of application controllers (daemons) that run on each node of the system, and a centralized controller is responsible for managing the execution of the distributed application.

Along the same lines, Weevil [37] automates the creation of deployment scripts. A set of models is provided by the user to describe the experiment. An interesting feature of Weevil lies in its ability to replay a distributed workload (such as a set of request for a distributed middleware infrastructure). These inputs can either be synthetically generated, or recorded from a previous run or simulation. The deployment phase does not include any node selection mechanism: the set of nodes and the mapping of application instances to these nodes must be provided by the user. The created scripts allow deployment and removal of the application, as well as the retrieval of outputs at the end of an experiment.

Plush and Weevil share a set of limitations that make them unsuitable for our goals. First, and most importantly, these systems propose high-end features for experienced users on experimental platforms such as PlanetLab, but cannot provide resource isolation due to their script-based nature. This restricts their usage to controlled testbeds, i.e., platforms on which the user has been granted some access rights, as opposed to non-dedicated environments such as networks of idle workstations where it might not be desirable or possible to create accounts on the machines, and where the nature of the testbed imposes to restrict the usage of their resources (e.g., disk or network usage).Second, they do not provide any management of the dynamics (churn) of the system, despite its recognized usefulness [29] for distributed system evaluation.

Testbeds. A set of experimental platforms, hereafter denoted as *testbeds*, have been built and proposed to the community. These testbeds are complementary to the languages and deployment systems presented in the first part of this section: they are the *medium* on which these tools operate.

Distributed simulation platforms such as WiDS [25] allow developers to run their application on top of an event-based network simulation layer. Distributed simulation is known to scale poorly, due to the high load of synchronization between nodes of the testbed hosting communicating processes. WiDS alleviates this limitation by relaxing the synchronization model between processes on distinct nodes. Nonetheless, event-based simulation testbeds such as WiDS do not provide mechanisms to deploy or manage the distributed application under test.

Network emulators such as Emulab [38], ModelNet [35], FlexLab [30] or P2PLab [28] can reproduce some of the characteristics of a networked environment: delays, bandwidth, packet drops, etc. They basically allow users to evaluate unmodified applications across various network models. Applications are typically deployed in a local-area cluster and all communications are routed through some proxy node(s), which emulate the topology. Each machine in the cluster can host several end-nodes from the emulated topology.

The PlanetLab [11] testbed (and forks such as Everlab [22]) allows experimenting in live networks by hosting applications on a large set of geographically dispersed hosts. It is a very valuable infrastructure for testing distributed applications in the most adverse conditions.

SPLAY is designed to complement these systems. Testbeds are useful but, often, complex platforms. They require the user to know how to deploy applications, to have a good understanding of the target topology, and to be able to properly configure the environment for executing his/her application (for instance, one needs to use a specific library to override the IP address used by the application in a ModelNet cluster). In PlanetLab, it is time-consuming and error-prone to choose a set of non-overloaded nodes on which to test the application, to deploy and launch the program, and to retrieve the results. Finally, considering mixed deployments that use several testbeds at the same time for a single experiment would require to write even much more complex scripts (e.g., taking into account problems such as port range forwarding). With SPLAY, as soon as the administrator who deployed the infrastructure has set up the network, using a complex testbed is as straightforward for the user as running an application on a local machine.

3 The SPLAY Framework

We present the architecture of our system: its main components, its programming language, libraries and tools.

3.1 Architecture

The SPLAY framework consists of about 15,000 lines of code written in C, Lua, Ruby, and SQL, plus some third-party support libraries. Roughly speaking, the architecture is made of three major components. These components are depicted in Figure 1.

- The **controller**, `splayctl`, is a trusted entity that controls the deployment and execution of applications.
- A lightweight **daemon** process, `splayd`, runs on every machine of the testbed. A `splayd` instantiates, stops, and monitors SPLAY applications when instructed by the controller.
- SPLAY **applications** execute in sandboxed processes forked by `splayd` daemons on participating hosts.

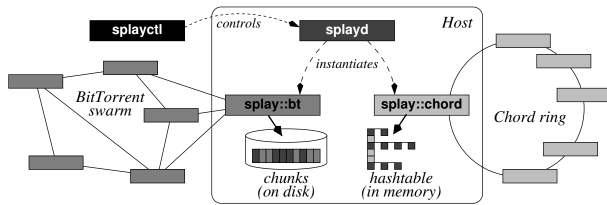


Figure 1: An illustration of two SPLAY applications (BitTorrent and Chord) at runtime.

Many SPLAY applications can run simultaneously on the same host. The testbed can be used transparently by multiple users deploying different applications on overlapping sets of nodes, unless the controller has been configured for a single-user testbed. Two SPLAY applications on the same node are unaware of each other (they cannot even exchange data via the file system); they can only communicate by message passing as for remote processes. Figure 1 illustrates the deployment of multiple applications with a host participating to both a Chord DHT and a BitTorrent swarm.

An important point is that SPLAY applications can be run locally with no modification to their code, while still using all libraries and language features proposed by SPLAY. Users can simply and quickly debug and test their programs locally, prior to deployment.

We now discuss in more details the different components of the SPLAY architecture.

Controller. The controller plays an essential role in our system. It is implemented as a set of cooperating processes and executes on one or several trusted servers. The only central component is a database that stores all data pertaining to participating hosts and applications.

The controller (see Figure 2) keeps track of all active SPLAY daemons and applications in the system. Upon startup, a daemon initiates a secure connection (SSL) to a `ctl` process. For scalability reasons, there can be many `ctl` processes spread across several trusted hosts. These processes only need to access the shared database.

SPLAY daemons open connections to `log` processes on behalf of the applications, if the logging library is used. This library is described in section 3.4.

The deployment of a distributed application is achieved by submitting a job through a command-line or Web-based interface. SPLAY also provides a Web services API that can be used by other projects. Once registered in the database, jobs are handled by `jobs` processes. The nodes participating in the deployment can be specified explicitly as a list of hosts, or one can simply indicate the number of nodes on which deployment has to take place, regardless of their identity. One can also specify requirements in terms of resources that must be available at the participating nodes (e.g., bandwidth) or in terms of geographical location (e.g., nodes in a specific country or within a given distance from a position). Incremental deployment, i.e., adding nodes at different times, can be performed using several jobs or with the churn manager.

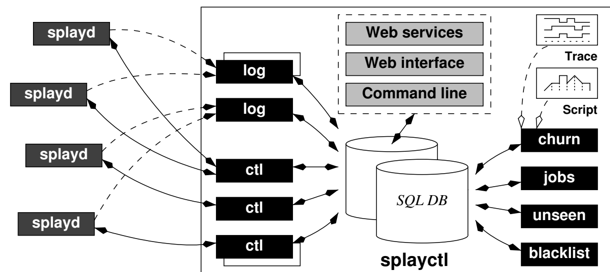
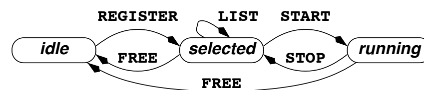


Figure 2: Architecture of the SPLAY controller (note that all components may be distributed on different machines).

Each daemon is associated with records in the database that store information about the applications and active hosts running them, or scheduled for execution. The controller monitors the daemons and uses a session mechanism to tolerate short-term disconnections (i.e., a daemon is considered alive if it shows activity at least once during a given time period). Only after a long-term disconnection (typically one hour) does the controller reset the status of the daemon and clean up the associated entries in the database. This task is under the responsibility of the `unseen` process. The `blacklist` process manages in the database a list of forbidden network addresses and masks; it piggybacks updates of this list onto messages sent to connected daemons.

Communication between the daemon and the controller follows a simple request/answer protocol. The first request originates from the daemon that connects to the controller. Every subsequent command comes from the controller. For brevity, we only present here a minimal set of commands.

The `jobs` process dequeues jobs from the database and searches for a set of hosts matching the constraints specified by the user. The controller sends a `REGISTER` message to the daemons of every selected node. In case the identity of the nodes is not explicitly specified, the system selects a set larger than the one originally requested to account for failed or overloaded nodes. Upon accepting the job, a daemon sends to the controller the range of ports that are available to the application. Once it receives enough replies, the controller first sends to every selected daemon a `LIST` message with the addresses of some participating nodes (e.g., a single *rendez-vous* node or a random subset , depending on the application) to bootstrap the application, followed by a `START` message to begin execution. Supernumerary daemons that are slow to answer and active applications that must be terminated receive a `FREE` message. The state machine of a SPLAY job is as follows:



The reason why we initially select a larger set of nodes than requested clearly appears when considering the availability of hosts on testbeds like PlanetLab, where

transient failures and overloads are the norm rather than the exception. Figure 3 shows both the cumulative and discretized distributions of round-trip times (RTT) for a 20 KB message over an already established TCP connection from the controller to PlanetLab hosts. One can observe that only 17.10% of the nodes reply within 250 milliseconds, and over 45% need more than 1 second. Selecting a larger set of candidates allows us to choose the most responsive nodes for deploying the application.

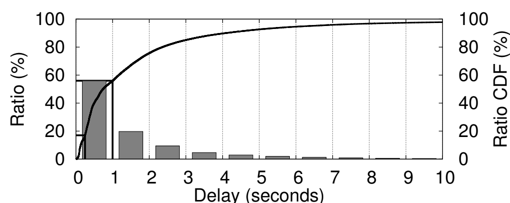


Figure 3: RTT between the controller and PlanetLab hosts over pre-established TCP connections, with a 20 KB payload.

Daemons. SPLAY daemons are installed on participating hosts by a local user or administrator. The local administrator can configure the daemon via a configuration file, specifying various instance parameters (e.g., daemon name, access key, etc.) and restrictions on the resources available for SPLAY applications. These restrictions encompass memory, network, and disk usage. If an application exceeds these limitations, it is killed (memory usage) or I/O operations fail (disk or network usage). The controller can specify stricter—but not weaker—restrictions at deployment time.

Upon startup, a SPLAY daemon receives a blacklist of forbidden addresses expressed as IP or DNS masks. By default, the addresses of the controllers are blacklisted so that applications cannot actively connect to them. Blacklists can be updated by the controller at runtime (e.g., when adding a new daemon or for protecting a particular machine).

The daemon also receives the address of a `log` process to connect to for logging, together with a unique identification key. SPLAY applications instantiated by the local daemon can only connect to that log process; other processes will reject any connection request.

3.2 Churn Management

In order to fully understand the behavior and robustness of a distributed protocol, it is necessary to evaluate it under different churn conditions. Theses conditions can range from rare but unpredictable hardware failures, to frequent application-level disconnections, as usually found in user-driven peer-to-peer systems, or even to massive failures scenarios. It is also important to allow comparison of competing algorithms under the very same churn scenarios. Relying on the natural, non-reproducible churn of testbeds such as PlanetLab often proves to be insufficient.

There exist several characterizations of churn that can be leveraged to reproduce realistic conditions for the protocol under test. First, synthetic descriptions issued from analytical studies [27] can be used to generate churn scenarios and replay them in the system. Second, several traces of the dynamics of real networks have been made publicly available by the community (e.g., see the repository at [1]); they cover a wide range of applications such as a highly churned file-sharing system [12] or high-performance computing clusters [32].

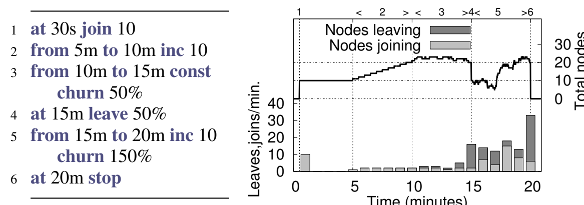


Figure 4: Example of a synthetic churn description: script (left), binned number of joins/leave (right, bottom) and total number of nodes (right, top).

SPLAY incorporates a component, `churn` (see Figure 2), dedicated to churn management. This component can send instructions to the daemons for stopping and starting processes on-the-fly. Churn can be specified as a trace, in a format similar to that used by [1], or as a synthetic description written in a simple script language. The trace indicates explicitly when each node enters or leaves the system while the script allows users to express phases of the application's lifetime, such as a steady increase or decrease of the number of peers over a given time duration, periods with continuous churn, massive failures, join flash crowds, etc. An example script is shown in Figure 4 together with a representation of the evolution of the node population and the number of arrivals and departures during each one-minute period: an initial set of nodes joins after 30 seconds, then the system stabilizes before a regular increase, a period with a constant population but a churn that sees half of the nodes leave and an equal number join, a massive failure of half of the nodes, another increase under high churn, and finally the departure of all the nodes.

Section 5.5 presents typical uses of the churn management mechanism in the evaluation of a large-scale distributed system. It is noteworthy that the churn management system relieves the need for fault injection systems such as Loki [16]. Another typical use of the churn management system is for long-running applications, e.g., a DHT that serves as a substrate for some other distributed application under test and needs to stay available for the whole duration of the experiments. In such a scenario, one can ask the churn manager to maintain a fixed-size population of nodes and to automatically bootstrap new ones as faults occur in the testbed.

3.3 Language and Applications

SPLAY applications are written in the Lua language [20], whose features are extended by SPLAY's libraries. This design choice was dictated by four majors factors. First,

the most important reason is that Lua has unique features that allow to simply and efficiently implement sandboxing. As mentioned earlier, sandboxing is a sound basis for execution in non-dedicated environments, where resources need to be constrained and where the hosting operating system must be shielded from possibly buggy or ill-behaved code. Second, one of SPLAY's goals is to support large numbers of processes within a single host of the testbed. This calls for a low footprint for both the daemons and the associated libraries. This excludes languages such as Java that require several megabytes of memory just for their execution environment. Third, SPLAY must ensure that the achieved performance is as good as the host system permits, and the features offered to the distributed system designer shall not interfere with the performance of the application. Fourth, SPLAY allows deployment of applications on any hardware and on any operating systems. This requires a "write-once, run everywhere" approach that calls for either an interpreted or bytecode-based language. Lua's unique features allow us to meet these goals of lightness, simplicity, performance, security and generality.

Lua was designed from the ground up to be an efficient scripting language with very low footprint. According to recent benchmarks [2], Lua is among the fastest interpreted scripting languages. It is reflective, imperative, and procedural with extensible semantics. Lua is dynamically typed and has automatic memory management with incremental garbage collection. The small footprint from Lua results from its design that provides flexible and extensible meta-features, rather than a complete set of general-purpose facilities. The full interpreter is less than 200 kB and can be easily embedded. Applications can use libraries written in different languages (especially C/C++). This allows for low-level programming if need be. Our experiments (Section 5) highlight the lightness of SPLAY applications using Lua, in terms of memory footprint, load, and scalability.

Lua's interpreter can directly execute source code, as well as hardware-dependent (but operating system-independent) bytecode. In SPLAY, the favored way of submitting applications is in the form of source code, but bytecode programs are also supported (e.g., for intellectual property protection).

Isolation and sandboxing are achieved thanks to Lua's support for first-class functions with lexical scoping and closures, which allow us to restrict access to I/O and networking libraries. We modify the behavior of these functions to implement the restrictions imposed by the administrator or by the user at the time he/she submits the application for deployment over SPLAY.

Lua also supports cooperative multitasking by the means of coroutines, which are at the core of SPLAY's event-based model (discussed below).

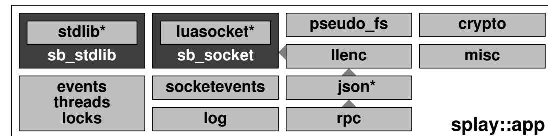


Figure 5: Overview of the main SPLAY libraries.

3.4 The Libraries

SPLAY includes an extensible set of shared libraries (see Figure 5) tailored for the development of distributed applications and overlays. These libraries are meant to be also used outside of the deployment system, when developing the application. We briefly describe the major components of these libraries.

Networking. The `luasocket` library provides basic networking facilities. We have wrapped it into a restricted socket library, `sb_socket`, which includes a security layer that can be controlled by the local administrator (the person who has instantiated the local daemon process) and further restricted remotely by the controller. This secure layer allows us to limit: (1) the total bandwidth available for SPLAY applications (instantaneous bandwidth can be limited using shaping tools if need be); (2) the maximum number of sockets used by an application and (3) the addresses that an application can or cannot connect to. Restrictions are specified declaratively in configuration files by the local user that starts the daemon, or at the controller via the command-line and Web-based APIs.

We have implemented higher-level abstractions for simplifying communication between remote processes. Our API supports message passing over TCP and UDP, as well as access to remote function and variables using RPCs. Calling a remote function is almost as simple as calling a local one (see code in next section). All arguments and return values are transparently serialized. Communication errors are reported using a second return value, as allowed by Lua.

Finally, communication libraries can be instructed to drop a given proportion of the packets (specified upon deployment): this can be used to simulate lossy links and study their impact on an application.

Sandboxed virtual filesystem. Overlays and distributed applications often need to use the local file system. For instance, when instantiating the BitTorrent protocol to replicate a large file on a set of nodes, temporary data must be written to disk as chunks are being received. Following our goal to not impact the hosting operating system, we need to ensure that a SPLAY application cannot access or overwrite any data on the host file system. To this end, SPLAY includes a library, `sb_fs`, that wraps the standard `io` library and provides restricted access to the file system in an OS-independent fashion.

Our wrapped library simulates a file system inside a single directory. The library transparently maps a com-

plete path name to the underlying files that stores the actual data, and applications can only read the files located in their private directory. The wrapped file handles enforce additional restrictions, such as limitations on the disk space and the number of opened files.

Events, threads and locks. SPLAY proposes a threading model based on Lua's coroutines combined with event-based programming. Unlike preemptive threads, coroutines yield the processor to each other (cooperative multitasking). This happens at special points in base libraries, typically when performing an operation that may block (e.g., disk or network I/O). This is typically transparent to the application developer. Although a *single* SPLAY application will not benefit from a multicore processor, coroutines are preferable to system-level threads for two reasons: their portability and their recognized efficiency (low latency and high throughput) for programs that use many network connections (using either non-blocking or RPC-based programming), which is typical of distributed systems programming. Moreover, using a single process (at the operating system level) has a lower footprint, especially from a sandboxing perspective, and allows deploying more applications on each `splayd`.

Shared data accesses are also safer with coroutines, as race conditions can only occur if the current thread yields the processor. This requires, however, a good understanding of the behavior of the application (we illustrate a common pitfall in Section 4). SPLAY provides a lock library as a simple alternative to protect shared data from concurrent accesses by multiple coroutines.

We have also developed an event library, `events`, that controls the main execution loop of the application, the scheduler, the communication between coroutines, timeouts, as well as event generation, waiting, and reception. To integrate with the event library, we have wrapped the socket library to produce a non-blocking, coroutine-aware version `sb_socket`. All these layers are transparent to the SPLAY developer who only sees a restricted, non-blocking socket library.

Logging. An important objective of SPLAY is to be able to quickly prototype and experiment with distributed algorithms. To that end, one must be able to easily debug and collect statistics about the SPLAY application at runtime. The `log` library allows the developer to print information either locally (screen, file) or, more interestingly, send it over the network to a log collector managed by the controller. If need be, the amount of data sent to the log collector can be restricted by a `splayd`, as instructed by the controller. As with most log libraries, facilities are provided to manage different log levels and dynamically enable or disable logging.

Other libraries. SPLAY provides a few other libraries with facilities useful for developing distributed systems and applications. The `llenc` and `json` libraries [3] support automatic and efficient serialization of data to be sent to remote nodes over the network. We developed the first one, `llenc`, to simplify message passing over stream-oriented protocols (e.g., TCP). The library automatically performs message demarcation, computing buffer sizes and waiting for all packets of a message before delivery. It uses the `json` library to automate encoding of any type of data structures using a compact and standardized data-interchange format. The `crypto` library includes cryptographic functions for data encryption and decryption, secure hashing, signatures, etc. The `misc` library provides common containers, functions for format conversion, bit manipulation, high-precision timers and distributed synchronization.

The memory footprint of these libraries is remarkably small. The base size of a SPLAY application is less than 600 kB with all the abovementioned libraries loaded. It is easy for administrators to deploy additional third-party software with the daemons, in the form of libraries. Lua has been design to seamlessly interact with C/C++, and other languages that bind to C can be used as well. For instance, we successfully linked some Splay application code with a third-party video transcoding library in C, for experimenting with adaptive video multicast. Obviously, the administrator is responsible for providing sandboxing in these libraries if required.

4 Developing Applications with SPLAY

This section illustrates the development of an application for SPLAY. We use the well-known Chord overlay [33] for its familiarity to the community. As we will see, the specification of this overlay is remarkably concise and close to the pseudo-code found in the original paper. We have successfully deployed this implementation on a ModelNet cluster and PlanetLab; results are presented in Section 5.2. The goal here is to provide the reader with a complete chain of development, deployment, and monitoring of a well-known distributed application. Note that local testing and debugging is generally done outside of the deployment framework (but still, using SPLAY libraries).

Chord is a distributed hash table (DHT) that maps keys to nodes in a peer-to-peer infrastructure. Any node can use the DHT substrate to determine the current live node that is responsible for a given key. When joining the network, a node receives a unique identifier (typically by hashing its IP address and port number) that determines its position in the identifier space. Nodes are organized in a ring according to their identifiers, and every node is responsible for the keys that fall between itself (inclusive) and its predecessor (exclusive). In addition to keeping track of their successors and predecessors on the ring, each node maintains a "finger" table whose entries point to nodes at an exponentially increasing distance from the current node's position. More precisely, the ith entry of a node with identifier n designates the live node responsible for key $n + 2^i$. Note that the successor is effectively the first entry in the finger table.

```
4  function join(n0)                                         -- n0: some node in the ring
5    predecessor = nil
6    finger[1] = call(n0, {'find_successor', n.id})
7    call(finger[1], {'notify', n})
8  end
9  function stabilize()                          -- periodically verify n's successor
10   local x = call(finger[1], 'predecessor')
11   if x and between(x.id, n.id, finger[1].id, false, false) then
12     finger[1] = x                                                  -- new successor
13   end
14   call(finger[1], {'notify', n})
15 end
16 function notify(n0)                         -- n0 thinks it might be our predecessor
17   if not predecessor or between(n0.id, predecessor.id, n.id, false, false) then
18     predecessor = n0                                             -- new predecessor
19   end
20 end
21 function fix_fingers()                                             -- refresh fingers
22   refresh = (refresh % m) + 1                                  -- 1 ≤ refresh ≤ m
23   finger[refresh] = find_successor((n.id + 2^(refresh − 1)) % 2^m)
24 end
25 function check_predecessor()                      -- checks if predecessor has failed
26   if predecessor and not ping(predecessor) then
27     predecessor = nil
28   end
29 end
```

Listing 1: SPLAY code for Chord overlay (stabilization).

Listing 1 shows the code for the construction and maintenance of the Chord overlay. For clarity, we only show here the basic algorithm that was proposed in [33] (the reader can appreciate the similarity between this code and Figure 6 of the referenced paper).

Function `join()` allows a node to join the Chord ring. Only its successor is set: its predecessor and successor's predecessor will be updated as part of the stabilization process. Function `stabilize()` periodically verifies that a node is its own successor's predecessor and notifies the successor. SPLAY base library's `between` call determines the inclusion of a value in a range, on a ring. Function `notify()` tells a node that its predecessor might be incorrect. Function `fix_fingers()` iteratively refreshes fingers. Finally, function `check_predecessor()` periodically checks if a node's predecessor has failed.

These functions are identical in their behavior and very similar in their form to those published in [33]. Yet, they correspond to executable code that can be readily deployed. The implementation of Chord illustrates a subtle problem that occurs frequently when developing distributed applications from a high-level pseudo-code description: the reception of multiple messages may trigger concurrent operations that perform conflicting modifications on the state of the node. SPLAY's coroutine model alleviates this problem in some, but not all, situations. During the blocking call to `ping()` on line 26 of Listing 1, a remote call to `notify()` can update the predecessor, which may be erased on line 27 until the next remote call to `notify()`. This is not a major issue as it may only delay stabilization, not break consistency. It can be avoided by adding an extra check after the ping or, more generally, by using the locks provided by the SPLAY standard libraries (not shown here).

```
30 function find_successor(id)                               -- ask node to find id's successor
31   if between(id, n.id, finger[1].id, false, true)   -- inclusive for second bound
32     return finger[1]
33   end
34   local n0 = closest_preceding_node(id)
35   return call(n0, {'find_successor', id})
36 end
37 function closest_preceding_node(id)                          -- finger preceding id
38   for i = m, 1, −1 do
39     if finger[i] and between(finger[i].id, n.id, id, false, false) then
40       return finger[i]
41     end
42   end
43   return n
44 end
```

Listing 2: SPLAY code for Chord overlay (lookup).

Listing 2 shows the code for Chord lookup. Function `find_successor()` looks for the successor of a given identifier, while function `closest_preceding_node()` returns the highest predecessor of a given identifier found in the finger table. Again, one can appreciate the similarity with the original pseudo-code.

This almost completes our minimal Chord implementation, with the exception of the initialization code shown in Listing 3. One can specifically note the registration of periodic stabilization tasks and the invocation of the main event loop.

```
1  require "splay.base"                       -- events, misc, socket (core libraries)
2  rpc = require "splay.rpc"                               -- rpc (optional library)
3  between, call, ping = misc.between_c, rpc.call, rpc.ping                -- aliases

45 timeout = 5                                               -- stabilization frequency
46 m = 24                          -- 2^m nodes and key with identifiers of length m
47 n = job.me                                                  -- our node {ip, port, id}
48 n.id = math.random(1, 2^m)                               -- random position on ring
49 predecessor = nil                               -- previous node on ring {id, ip, port}
50 finger = {[1] = n}                                   -- finger table with m entries
51 refresh = 0                                                  -- next finger to refresh
52 n0 = job.nodes[1]                                   -- first peer is rendez−vous node
53 rpc.server(n.port)                                              -- start rpc server
54 events.thread(function() join(n0) end)                           -- join chord ring
55 events.periodic(stabilize, timeout)               -- periodically check successor, ...
56 events.periodic(check_predecessor, timeout)                        -- predecessor, ...
57 events.periodic(fix_fingers, timeout)                                  -- and fingers
58 events.loop()                                                     -- execute main loop
```

Listing 3: SPLAY code for Chord overlay (initialization).

While this code is quite classical in its form, the remarkable features are the conciseness of the implementation, the closeness to pseudo-code, and the ease with which one can communicate with other nodes of the system by RPC. Of course, most of the complexity is hidden inside the SPLAY infrastructure.

The presented implementation is not fault-tolerant. Although the goal of this paper is not to present the design of a fault-tolerant Chord, we briefly elaborate below on some steps needed to make Chord robust enough for running on error-prone platforms such as PlanetLab. The first step is to take into account the absence of a reply to an RPC. Consider the call to `predecessor` in method `stabilize()`. One simply needs to replace this call by the code of Figure 4.

```
1 function stabilize()                    -- rpc.a_call() returns both status and results
2   local ok, x = rpc.a_call(finger[1], 'predecessor', 60)    -- RPC, 1m timeout
3   if not ok then
4     suspect(finger[1])          -- will prune the node out of local routing tables
5   else
6     (...)
```

Listing 4: Fault-tolerant RPC call

We omit the code of function `suspect()` for brevity. Depending on the reliability of the links, this function prunes the suspected node after a configurable number of missed replies. One can tune the RPC timeout according to the target platform (here, 1 minute instead of the standard 2 minutes), or use an adaptive strategy (e.g., exponentially increasing timeouts). Finally, as suggested by [33] and similarly to the leafset structure used in Pastry [31], we replace the single successor and predecessor by a list of 4 peers in each direction on the ring.

Our Chord implementation without fault-tolerance is only 58 lines long, which represents an increase of 18% over the pseudo-code from the original paper (which does not contain initialization code, while our code does). Our fault-tolerant version is only 100 lines long, i.e., 73% more than the base implementation (29% for fault tolerance, and 44% for the leafset-like structure). We detail the procedure for deployment and the results obtained with both versions on a ModelNet cluster and on PlanetLab, respectively, in Section 5.2.

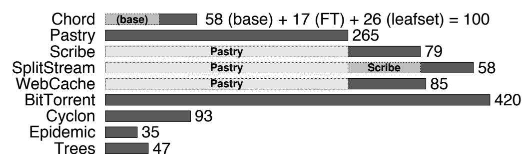
5 Evaluation

This section presents a thorough evaluation of SPLAY performance and capabilities. Evaluating such an infrastructure is a challenging task as the way users will use it plays an important role. Therefore, our goal in this evaluation is twofold: (1) to present the implementation, deployment and observation of real distributed systems by using SPLAY's capability to easily reproduce experiments that are commonly used in evaluations and (2) to study the performance of SPLAY itself, both by comparing it to other widely-used implementations and by evaluating its costs and scalability. The overall objective is to demonstrate the usefulness and benefits of SPLAY rather than evaluate the distributed applications themselves. We first demonstrate in Section 5.1 SPLAY's capabilities to easily express complex system in a concise manner. We present in Section 5.2 the deployment and performance evaluation of the Chord DHT proposed in Section 4, using a ModelNet [35] cluster and PlanetLab [11]. We then compare in Section 5.3 the performance and scalability of the Pastry [31] DHT written with SPLAY against a legacy Java implementation, FreePastry [4]. Sections 5.4 and 5.5 evaluate SPLAY's ability to easily (1) deploy applications in complex network settings (mixed PlanetLab and ModelNet deployment) and (2) reproduce arbitrary churn conditions. Section 5.6 focuses on SPLAY performance for deploying and undeploying applications on a testbed. We conclude in Section 5.7 with an evaluation of SPLAY's performance with resource-intensive applications (tree-based content dissemination and long-term running of a cooperative Web cache).

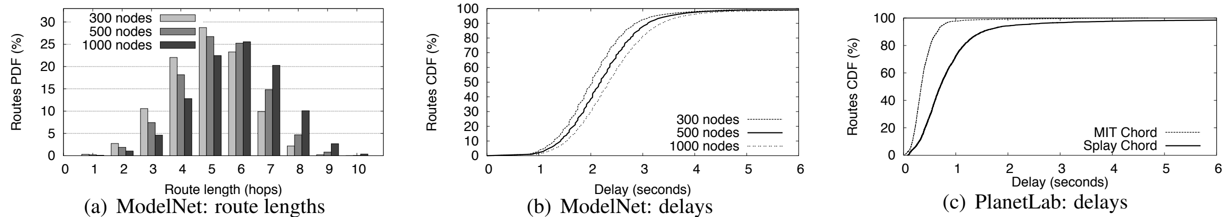
Experimental setup. Unless specified otherwise, our experimentations were performed either on PlanetLab, using a set of 400 to 450 hosts, or on our local cluster (11 nodes, each equipped with a 2.13 Ghz Core 2 Duo processor and 2 GB of memory, linked by a 1 Gbps switched network). All nodes run GNU/Linux 2.6.9. A separate node running FreeBSD 4.11 is used as a ModelNet router, when required by the experiment. Our ModelNet configuration emulates 1,100 hosts connected to a 500-node transit-stub topology. The bandwidth is set to 10Mbps for all links. RTT between nodes of the same domain is 10 ms, stub-stub and stub-transit RTT is 30 ms, and transit-transit (i.e., long range links) RTT is 100 ms. These settings result in delays that are approximately twice those experienced in PlanetLab.

5.1 Development complexity

We developed the following applications using SPLAY: Chord [33] and Pastry [31], two DHTs; Scribe [15], a publish-subscribe system; SplitStream [14], a bandwidth-intensive multicast protocol; a cooperative web-cache based on Pastry; BitTorrent [17], a content distribution infrastructure;[1] and Cyclon [36], a gossip-based membership management protocol. We have also implemented a number of classical algorithms, such as epidemic diffusion on Erdös-Renyi random graphs [19] and various types of distribution trees [13] (n-ary trees, parallel trees). As one can note from the following figure, all implementations are extremely concise in terms of lines of code (LOC). Note that we did not try to compact the code in a way that would impair readability. Numbers and darker bars represent LOC for the protocol, while lighter bars represent protocols acting as a substrate (Scribe and our Web cache are based on Pastry, SplitStream is based on both Pastry and Scribe):



Although the number of lines is clearly just a rough indicator of the expressiveness of a system, it is still a valuable metric to estimate programming efforts. Our implementations are systematically more compact than those written with Mace [23] (by approximately a factor of two) and comparable to P2's [26] specifications. A well-documented protocol such as Chord only took a few hours to implement and debug. In contract, BitTorrent, being a complex and underspecified protocol, required several days of development. In both cases, the development process greatly benefited from the short deployment and testing phase, made almost trivial by SPLAY.



(a) ModelNet: route lengths (b) ModelNet: delays (c) PlanetLab: delays

Figure 6: Performance results of Chord, deployed on a ModelNet cluster and on PlanetLab.

5.2 Testing the Chord Implementation

This section presents the deployment and performance results of the Chord implementation from Section 4. We proceed with two deployments. First, the exact code presented in this paper is deployed in a ModelNet testbed with no node failure. Second, a slightly modified version of this code is run on PlanetLab. This version includes the extensions presented at the end of Section 4: use of a leaf set instead of a single successor and a single predecessor, fault-tolerant RPCs, and shorter stabilization intervals.

Chord on ModelNet. To parameterize the deployment of the Chord implementation presented in Section 4 on a testbed, we create a descriptor that describes resources requirements and limitations. The descriptor allows to further restrict memory, disk and network usage, and it specifies what information an application should receive when instantiated:

```
--[[ BEGIN SPLAY RESOURCES RESERVATION
  nb_splayd  1000
  nodes      head    1
END SPLAY RESOURCES RESERVATION ]]
```

This descriptor requests 1,000 instances of the application and specifies that each instance will receive three essential pieces of information: (1) a single-element list containing the first node in the deployment sequence (to act as rendezvous node); (2) the rank of the current process in the deployment sequence; and (3) the identity of the current process (host and port). This information is useful to bootstrap the system without having to rely on external mechanisms such as a directory service. In the case of Chord, we use this information to have hosts join the network one after the other, with a delay between consecutive joins to ensure that a single ring is created. A staggered join strategy allows better experiments reproducibility, but a massive join scenario would succeed as well. The following code is added:

```
events.sleep(job.position)          -- 1s between joins
if #job.position > 1 then           -- first node is rendez-vous node
  join(job.nodes[1])
end
```

Finally, we register the Lua script and the deployment descriptor using one of the command line, Web service or Web-based interfaces.

Each host runs 27 to 91 Chord nodes (we show in Section 5.3 that SPLAY can handle many more instances on a single host). During the experiment, each node injects 50 random lookup requests in the system. We then undeploy the overlay, and process the results obtained from the logging facility. Figure 6(a) presents the distribution of route lengths. Figure 6(b) presents the cumulative distribution of latencies. The average number of hops is below $\frac{\log_2 N}{2}$ and the look-up time remains small. This supports our observations that SPLAY is efficient and does not introduce additional delays or overheads.

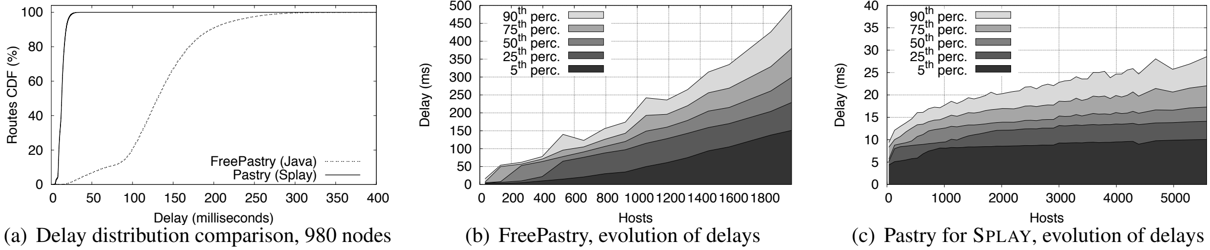
Chord on PlanetLab. Next, we deploy our Chord implementation with extensions on 380 PlanetLab nodes and compare its performance with MIT's fine-tuned C++ Chord implementation [5] in terms of delays when looking up random keys in the DHT. In both cases, we let the Chord overlay stabilizes before starting the measurements. Figure 6(c) presents the cumulative distribution of delays for 5000 random lookups (average route length is 4.1 for both systems). We observe that MIT Chord outperforms Chord for SPLAY, because it relies on a custom network layer that uses, amongst other optimizations, network coordinates for constructing latency-aware finger tables. In contrast, we did not include such optimizations in our implementation.

5.3 SPLAY Performance

We evaluate the performance of applications using SPLAY in two ways. First, we evaluate the efficiency of the network libraries, based on the delays experienced by a sample application on a high-performance testbed. Second, we evaluate scalability: how many nodes can be run on a single host and what is the impact on performance. For these tests we chose Pastry [31] because: (i) it combines both TCP and UDP communications; (ii) it requires efficient network libraries and transport layers, each node being potentially opening sockets and sending data to a large number of other peers; (iii) it supports network proximity-based peer selection, and as such can be affected by fluctuating or unstable delays (for instance due to overload or scheduling issues).

We compare our version of Pastry with FreePastry 2.0 [4], a complete implementation of the Pastry protocol in Java. Our implementation is functionally identical to FreePastry and uses the very same protocols, e.g., locality-aware routing table construction and stabilization mechanisms to repair broken routing table entries. The only notable differences reside in the message formats (no wire compatibility) and the choice of alternate routes upon failure.

We deployed FreePastry using all optimizations ad-



(a) Delay distribution comparison, 980 nodes (b) FreePastry, evolution of delays (c) Pastry for SPLAY, evolution of delays

Figure 7: Comparisons of two implementations of Pastry: FreePastry and Pastry for SPLAY.

vised by the authors, that is, running multiple nodes within the same JVM, replacing Java serialization with raw serialization, and keeping a pool of opened TCP connections to peers to avoid reopening recently used connections. We used 3 JVMs on our dual cores machines, each running multiple Pastry nodes. With a large set of nodes, our experiments have shown that this configuration yields slightly better results than using a single JVM, both in terms of delay and load.

Figure 7(a) presents the cumulative delay distribution in a converged Pastry ring. The distribution of route lengths (not shown) is slightly better with FreePastry thanks to optimizations in the routing table management. Delays obtained with Pastry on SPLAY are much lower than the delays obtained with FreePastry. This experiment shows that SPLAY, while allowing for concise and readable protocol implementations, does not trade simplicity for efficiency. We also notice that Java-based programs are often too heavyweight to be used with multiple instances on a single host.[2] This is further conveyed by our second experiment that compares the evolution of delays of FreePastry (Figure 7(b)) and Pastry for SPLAY (Figure 7(c)) as the number of nodes on the testbed increases. We use a percentile-based plotting method that allows expressing the evolution of a cumulative distribution of delays with respect to the number of nodes. We can observe that: (1) delays start increasing exponentially for FreePastry when there are more than 1,600 nodes running in the cluster, that is 145 nodes per host (recall that all nodes on a single host are hosted by only 3 JVMs and share most of their memory footprint); (2) it is not possible to run more than 1,980 FreePastry nodes, as the system will start swapping, degrading performance dramatically; (3) SPLAY can handle 5,500 nodes (500 on each host) without significant drop in performance (other than the $\mathcal{O}(\log N)$ route lengths evolution, N being the number of nodes).

Figure 8 presents the load (i.e., average number of processes with "runnable" status, as reported by the Linux scheduler) and memory consumption per instance for varying number of instances. Each process is a Pastry node and issues a random request every minute. We observe that the memory footprint of an instance is lower than 1.5 MB, with just a slight increase during the experiment as nodes fill their routing table. It takes 1,263 Pastry instances before the host system starts swapping memory to disk. Load (averaged over the last minute) remains reasonably low, which explains the small delays presented by Figure 7(c).

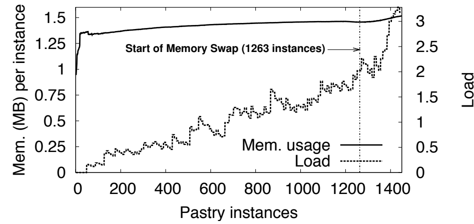


Figure 8: Memory consumption and load evolution on a single node hosting several instances of Pastry for SPLAY.

5.4 Complex Deployments

SPLAY is designed to be used within a large set of different testbeds. Despite this diversity, it is sometimes also desirable to experiment with more than a single testbed at a time. For instance, one may want to evaluate a complex system with a set of peers linked by high bandwidth, non-lossy links, emulated by ModelNet, and a set of peers facing adverse network conditions on PlanetLab. A typical usage would be to test a broker-based publish-subscribe infrastructure deployed on reliable nodes, along with a set of client nodes facing churn and lossy network links.

Such a mixed deployment requires a deep understanding of the system for setting it up using scripting and common tools, as the user has to care about NAT and firewalls traversal, port forwarding, etc. The experiment presented in this section shows that such a complex mixed deployment can be achieved using SPLAY as if it were on a single testbed. The only precondition is that the administrator of the part of the testbed that is behind a NAT or firewall defines (and opens) a range of ports that all `splayds` will use to communicate with other daemons outside the testbed. Notably for a ModelNet cluster, this operation can easily be done at the time Modelnet is installed on the nodes of the testbed and it does not requires additional access rights. All other communication details are dealt with by SPLAY itself: no modification is needed to the application code.

Figure 9 presents the delay distribution for a deployment of 1,000 nodes on PlanetLab, on ModelNet, and in a mixed deployment over both testbeds at the same time (i.e., 500 nodes on each). We notice that the delays of the mixed deployment are distributed between the delays of

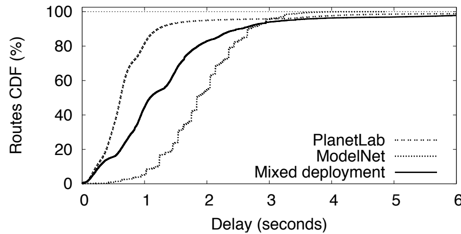


Figure 9: Pastry on PlanetLab, ModelNet, and both.

PlanetLab and the higher delays of our ModelNet cluster. The "steps" on the ModelNet cumulative delays representation are a result of routes of increasing number of hops (both in Pastry and in the emulated topology), and the fixed delays for ModelNet links.

5.5 Using Churn Management

This section evaluates the use of the churn management module, both using traces and synthetic descriptions. Using churn is as simple as launching a regular SPLAY application with a trace file as extra argument. SPLAY provides a set of tools to generate and process trace files. One can, for instance, speed-up a trace, increase the churn amplitude whilst keeping its statistical properties, or generate a trace from a synthetic description.

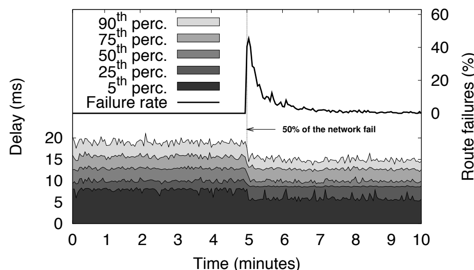


Figure 10: Using churn management to reproduce massive churn conditions for the SPLAY Pastry implementation.

Figure 10 presents a typical experiment of a massive failure using the synthetic description. We ran Pastry on our local cluster with 1,500 nodes and, after 5 minutes, triggered a sudden failure of half of the network (750 nodes). This models, for example, the disconnection of a inter-continental link or a WAN link between two corporate LANs. We observe that the number of failed lookups reaches almost 50% after the massive failure due to routing table entries referring to unreachable nodes. Pastry recovers all its routing capabilities in about 5 minutes and we can observe that delays actually decrease after the failure because the population has shrunk (delays are shown for successful routes only). While this scenario is amongst the simplest ones, churn descriptions allow users to experiment with much more complex scenarios, as discussed in Section 3.2.

Our second experiment is representative of a complex test scenario that would usually involve much engineering, testing and post-processing. We use the churn trace observed in the Overnet file sharing peer-to-peer system [12]. We want to observe the behavior of Pastry, deployed on PlanetLab, when facing churn rates that are much beyond the natural churn rates suffered in PlanetLab. As we want increasing levels of Churn, we simply "speed-up" the trace, that is, with a speed-up factor of 2x, 5x or 10, a minute in the original trace is mapped to 30, 12 or 6 seconds respectively. Figure 11 presents both the churn description and the evolution of delays and failure rates, for increasing levels of churn. The churn description shows the population of nodes and the number of joins/leaves as a function of time, and performance observations plot the evolution of the delay distribution as a function of time. We observe that (1) Pastry handles churn pretty well as we do not observe a significant failure rate when as much as 14% of the nodes are changing state within a single minute; (2) running this experiment is neither more complex nor longer than on a single cluster without churn, as we did for Figure 7(a). Based on our own experience, we estimate that it takes at least one order of magnitude less human efforts to conduct this experiment using SPLAY than with any other deployment tools. We strongly believe that the availability of tools such as SPLAY will encourage the community to further test and deploy their protocols under adverse conditions, and to compare systems using published churn models.

5.6 Deployment Performance

This section presents an evaluation of the deployment time of an application on an adversarial testbed, PlanetLab. This further conveys our position from Section 3.1 that one needs to initially select a larger set of nodes than requested to ensure that one can rely on reasonably responsive nodes for deploying the application. Traditionally, such a selection process is done by hand, or using simple heuristics based on the load or response time of the nodes. SPLAY relieves the need for the user to proceed with this selection. Figure 12 presents the deployment time for the Pastry application on PlanetLab. We vary the number of additionally probed daemons from 10% to 100% of the requested nodes. We observe that a larger set results in lower delays for deploying an application (hence, presumably, lower delays for subsequent application communications). Nonetheless, the selection of a reasonably large superset for a proper selection of peers is a tradeoff between deployment delay and redundant messages sent over the network. Based on experiments, we use by default an initial superset of 125% of requested nodes.

5.7 Resource-intensive Experiments

Our two last experimental demonstrations deal with resource-intensive applications, both for short-term and long-term runs. They further conveys SPLAY's ability to run in high performance settings and production environments, as well as demonstrating that the obtained performance is similar to the one achieved with a dedicated implementation (particularly from the network point of view). We run the following two experiments: (1) the

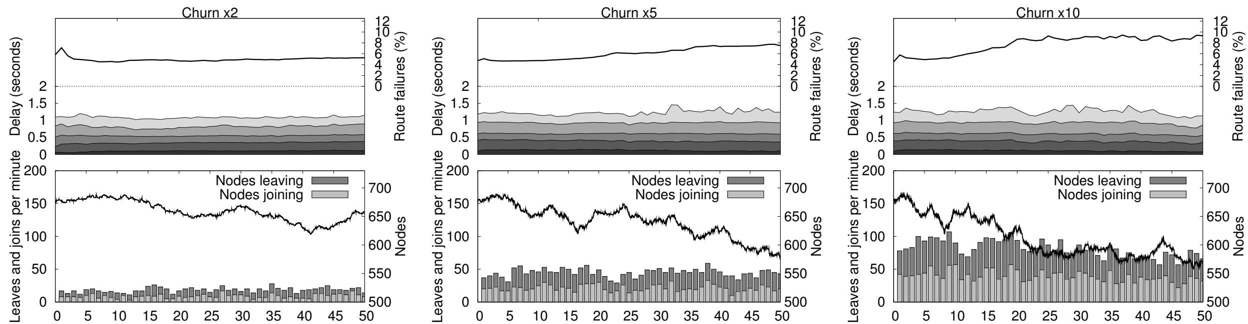


Figure 11: Study of the effect of churn on Pastry deployed on PlanetLab. Churn is derived from the trace of the Overnet file sharing system and sped up for increasing volatility.

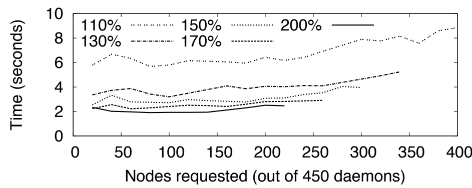


Figure 12: Deployment times of Pastry for SPLAY, as a function of (1) the number of nodes requested and (2) the size of the superset of daemons used.

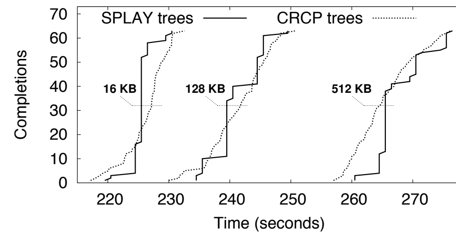


Figure 13: File distribution using trees.

evaluation of a cooperative data distribution algorithm based on parallel trees using both SPLAY and a native (C) implementation on ModelNet and (2) a distributed cooperative Web cache for HTTP accesses, which has been running for several weeks under a constant and significant load.

Dissemination using trees. This experiment compares two versions of a simple cooperative protocol [13] based on parallel n-ary trees written with SPLAY and in C. We create $n = 2$ distinct trees in the same manner as SplitStream [14] does: each of the 63 nodes is an inner member in one tree and a leaf in the other. The data to be transmitted is split into blocks, which are propagated along one of the 2 trees according to a round-robin policy. This experiment allows us to observe how SPLAY compares against a native application, CRCP, written in C [6]. Using a tree for this comparison bears the advantage of highlighting the additional delays and overheads of the platform and its network libraries (such as the sandboxing of network operations). These overheads accumulate at each level of the tree, from the root to the leaves.

Tests were run in a ModelNet testbed configured with a symmetric bandwidth of 1 Mbps for each node. Results are shown in Figure 13 for binary trees, a 24 MB file, and different block sizes (16 KB, 128 KB, 512 KB). We observe that both implementations produce similar results, which tends to demonstrate that the overhead of SPLAY's language and libraries is negligible. Differences in shape are due to CRCP nodes sending chunks sequentially to their children, while SPLAY nodes send chunks in parallel. In our settings (i.e., homogeneous bandwidth), this should not change the completion time of the last peer as links are saturated at all times.

Long-term experiment: cooperative Web cache. Our last experiment presents the performance over time of a cooperative Web cache built using SPLAY following the same base design as Squirrel [21]. This experiment highlights the ability of SPLAY to support long-run applications under constant load. The cache uses our Pastry DHT implementation deployed in a cluster, with 100 nodes that proxy requests and store remote Web resources for speeding up subsequent accesses. For this experiment, we limit the number of entries stored by each nodes to 100. Cached resources are evicted according to an LRU policy or when they are older than 120 seconds. The cooperative Web cache has been run for three weeks. Figure 14 presents the evolution of HTTP requests delay distribution for a period of 100 hours along with the cache hit ratio. We injected a continuous stream of 100 requests per second extracted from real Web access traces [7] corresponding to 1.7 million hits to 42,000 different URLs. We observe a steady cache hit ratio of 77.6%. The experienced delays distribution has remained stable throughout the whole run of the application. Most accesses (75th percentile) are cached and served in less than 25 to 100 ms, compared to non-cached accesses that require 1 to 2 seconds on average.

6 Conclusion

SPLAY is an infrastructure that aims at simplifying the development, deployment and evaluation of large-scale distributed applications. It incorporates several novel features not found in existing tools and testbeds. SPLAY applications are specified using in a high-level, efficient scripting language very close to pseudo-code commonly used by researchers in their publications. They execute

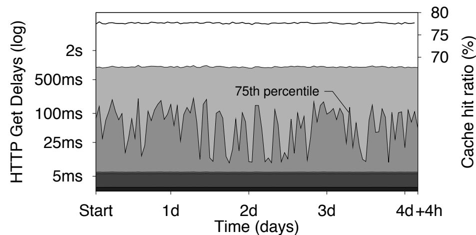


Figure 14: Cooperative Web cache: evolution of delays and cache hit ratios during a 4 days period.

in a sandboxed environment and can thus be readily deployed on non-dedicated hosts. SPLAY also includes a comprehensive set of shared libraries tailored for the development of distributed protocols. Application specifications are based on an event-driven model and are extremely concise.

SPLAY can seamlessly deploy applications in real (e.g., PlanetLab) or emulated (e.g., ModelNet) networks, as well as mixed environments. An original feature of SPLAY is its ability to inject churn in the system using a trace or a synthetic description to test applications in the most realistic conditions. Our thorough evaluation of SPLAY demonstrates that it allows developers to easily express complex systems in a concise yet readable manner, scales remarkably well thanks to its low footprint, exhibits very good performance in various deployment scenarios, and compares favorably against native applications in our experiments. SPLAY is publicly available from http://www.splay-project.org.

Acknowledgments: We would like to thank the anynymous reviewers for their constructive comments, as well as Petros Maniatis whose help was invaluable for preparing the final version of this paper.

References

[1] http://www.cs.berkeley.edu/~pbg/availability/.

[2] http://shootout.alioth.debian.org/.

[3] http://www.ietf.org/rfc/rfc4627.txt.

[4] http://freepastry.rice.edu/.

[5] http://pdos.csail.mit.edu/chord/.

[6] http://www.crossflux.org/crcp/.

[7] http://ftp.ircache.net/Traces/.

[8] ABBOTT, M., PETERSON, L. A language-based approach to protocol implementation. *IEEE/ACM Trans. Netw. 1*, 1 (Feb. 1993), 4–19.

[9] ALBRECHT, J., BRAUD, R., DAO, D., TOPILSKI, N., TUTTLE, C., SNOEREN, A. C., VAHDAT, A. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *LISA'07*.

[10] ANDERSON, D. Automated protocol implementation with rtag. *IEEE Trans. Soft. Eng. 14*, 3 (1988), 291–300.

[11] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., WAWRZONIAK, M. Operating system support for planetary-scale network services. In *NSDI'04*.

[12] BHAGWAN, R., SAVAGE, S., VOELKER, G. M. Understanding availability. In *IPTPS* (Feb. 2003).

[13] BIERSACK, E., RODRIGUEZ, P., FELBER, P. Performance analysis of peer-to-peer networks for file distribution. In *QofIS'04*.

[14] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., SINGH, A. SplitStream: High-bandwidth multicast in a cooperative environment. In *SOSP'03*.

[15] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., ROWSTRON, A. SCRIBE: A large-scale and decentralized publish-subscribe infrastructure. *IEEE J. Sel. Areas Commun. 20*, 8 (Oct. 2002).

[16] CHANDRA, R., LEFEVER, R. M., CUKIER, M., SANDERS, W. H. Loki: A state-driven fault injector for distributed systems. In *DSN'00*.

[17] COHEN, B. Incentives to build robustness in BitTorrent. Tech. rep., http://www.bittorrent.org/, May 2003.

[18] DUPUY, A., SCHWARTZ, J., YEMINI, Y., BACON, D. Nest: a network simulation and prototyping testbed. *Commun. ACM 33*, 10 (1990), 63–74.

[19] ERDÖS, P., RÉNYI, A. On the evolution of random graphs. *Mat. Kuttató. Int. Közl. 5* (1960), 17–60.

[20] IERUSALIMSCHY, R., DE FIGUEIREDO, L., CELES, W. The implementation of lua 5.0. *J. of Univ. Comp. Sc. 11*, 7 (2005), 1159–1176.

[21] IYER, S., ROWSTRON, A., DRUSCHEL, P. Squirrel: a decentralized peer-to-peer web cache. In *PODC'02*.

[22] JAFFE, E., BICKSON, D., KIRKPATRICK, S. Everlab: a production platform for research in network experimentation and computation. In *LISA'07*.

[23] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., VAHDAT, A. M. Mace: language support for building distributed systems. In *PLDI'07*.

[24] KOHLER, E., KAASHOEK, M., MONTGOMERY, D. A readable TCP in the prolac protocol language. *SIGCOMM Comput. Commun. Rev. 29*, 4 (1999).

[25] LIN, S., PAN, A., GUO, R., ZHANG, Z. Simulating large-scale P2P systems with the wids toolkit. In *MASCOTS'05*.

[26] LOO, B. T., CONDIE, T., HELLERSTEIN, J., MANIATIS, P., ROSCOE, T., STOICA, I. Implementing declarative overlays. In *SOSP'05*, pp. 75–90.

[27] MICKENS, J. W., NOBLE, B. D. Exploiting availability prediction in distributed systems. In *NSDI'06*.

[28] NUSSBAUM, L., RICHARD, O. Lightweight emulation to study peer-to-peer systems. *Concurr. Comput. : Pract. Exper. 20*, 6 (2008), 735–749.

[29] RHEA, S., GEELS, D., ROSCOE, T., KUBIATOWICZ, J. Handling churn in a dht. In *2004 USENIX Annual Technical Conference*.

[30] RICCI, R., DUERIG, J., SANAGA, P., GEBHARDT, D., HIBLER, M., ATKINSON, K., ZHANG, J., KASERA, S., LEPREAU, J. The Flexlab approach to realistic evaluation of networked systems. In *NSDI'07*.

[31] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware'01*.

[32] SCHROEDER, B., GIBSON, G. Large-scale study of failures in high-performance-computing systems. In *DSN'06*.

[33] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M., DABEK, F., BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1 (2003), 17–32.

[34] URBAN, P., DEFAGO, X., SCHIPER, A. Neko: A single environment to simulate and prototype distributed algorithms. In *ICOIN'01*.

[35] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., BECKER, D. Scalability and accuracy in a large-scale network emulator. In *OSDI'02*.

[36] VOULGARIS, S., GAVIDIA, D., VAN STEEN, M. CYCLON: Inexpensive membership management for unstructured P2P overlays. *J. Network Syst. Manage. 13*, 2 (2005).

[37] WANG, Y., CARZANIGA, A., WOLF, A. L. Four enhancements to automated distributed system experimentation methods. In *ICSE'08*.

[38] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI'02*.

Notes

[1]Note that, without the requirement for binary compatibility, the size of our implementation could be significantly reduced. Our BitTorrent implementation has been successfully used for downloading several times the Ubuntu Linux disk in official swarms.

[2]This possibility is notably useful to test characteristics that do not depend much on the performance of individual nodes with a limited-size testbed, e.g., to evaluate the scalability of routing in an overlay.