An overview of new features in the SPLAY framework for simple distributed systems evaluation

Lucas Charles Pascal Felber Raluca Halalai Etienne Rivière Valerio Schiavoni José Valerio

Institut d'Informatique, Université de Neuchâtel, Switzerland

http://www.splay-project.org, first.last@unine.ch

Abstract

Evaluating large-scale distributed applications is usually a complex, time-consuming and error-prone task. The use of experimental testbeds such as PlanetLab [4], ModelNet [26]-enabled clusters, Emulab [28]), or non-dedicated testbeds such as networks of workstations, is desirable for distributed systems and algorithms evaluation but often unfavored for simulations due to the complexity gap between algorithms specifications (pseudocode) and actual implementations.

SPLAY is a framework that simplifies the prototyping, development, deployment and evaluation of large-scale systems. The SPLAY system and its evaluation were presented two years ago at the NSDI conference [21]. Since then, new features and possibilities have been added to it, as well as performance, scalability and flexibility improvements. This technical report relates on these evolutions, and acts as a companion report to our NSDI paper for updated information on SPLAY. Documentation and further information can be found on the project's Web site. SPLAY is open source and can be downloaded from the same location: `http://www.splay-project.org`.

The novel features covered in this report include (1) user-space network topology emulation, (2) a novel distributed synchronization service, (3) improved management of churn replay from traces, (4) native library shipping for dynamically extending the libraries available on the testbed without re-deploying the persistent daemons and without accessing directly the testbed nodes, (5) improvements of the command-line interface, and (6) the possibility to use SPLAY as a batch submission service that is able to schedule and queue jobs. We conclude this technical report by some discussion of future work and directions.

Keywords SPLAY, distributed systems, distributed algorithms, evaluation, experimentation, deployment, language, runtime

1. Introduction

Developing large-scale distributed applications is complex, time-consuming and error-prone. The lack of appropriate tools for quickly prototyping, deploying and evaluating distributed algorithms in real settings introduces more difficulty to this task. Nonetheless, it is quite common to discover discrepancies between the modeled behavior of an application and its actual behavior when deployed in a live network. Actual evaluation on real conditions is therefore highly desirable and often unavoidable.

While there exist a number of experimental testbeds to address this demand (e.g., PlanetLab [4], ModelNet [26], or Emulab [28]), it is far from straightforward to develop, deploy, execute and monitor applications for and on these platforms and the learning curve is usually slow. The complexity of using existing testbeds discourages researchers, teachers, or systems practitioners from fully exploiting these technologies. Too often, they prefer relying on simulations, which do not always reflect accurately the reality of a real deployment.

These limitations are addressed by SPLAY, an infrastructure that simplifies the prototyping, development, deployment and evaluation of large-scale systems. 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 the Lua [17]. A comprehensive set of libraries allow SPLAY to extend Lua with a number of features that are useful for distributed systems development: RPC calls with transparent marshaling, support for cooperative multithreading, libraries for cryptography, security, network management, etc.

The development of SPLAY is a continuous task, and during the last 2 years several new features have been implemented, to reinforce the platform's usefulness, performance and flexibility. Most of these improvements have been driven by our group's research and evaluation activities, for instance in the context of context-aware distributed search [12, 13], privacy preservation and trust in large data stores [14], decentralized private group communication in large-scale autonomous systems [24], auditing and distributed certification mechanisms [27] or finally large-scale dissemination protocols [15, 22].

This technical report acts as a complement to our NSDI paper [21] published in 2009. The goal of this report is to present the novel features we added to the SPLAY system, their rationale, implementation, and evaluation of performance whenever applicable.

This report is structured as follows. In Section 2, we give an overview SPLAY as it was presented in our seminal paper [21]. In Section 3, we present the novel functionalities that we added to SPLAY, while Section 4 present our improvements to existing functionalities. In Section 5 we describe new features that we intend to add in the future, and conclude in Section 6.

2. An overview of SPLAY

In this section we present a short overview on the SPLAY architecture. Complete details can be found in our NSDI paper [21] as well as on our Web site `http://www.splay-project.org`.

Université de Neuchâtel Technical Report 2012-IIUN-1.

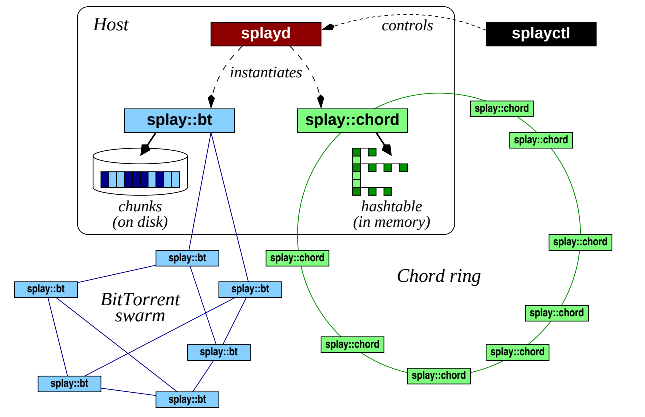

Figure 1. Example of the deployment of two applications using SPLAY on a testbed: a BitTorrent [8] file-sharing application, and the Chord [25] distributed hash table.

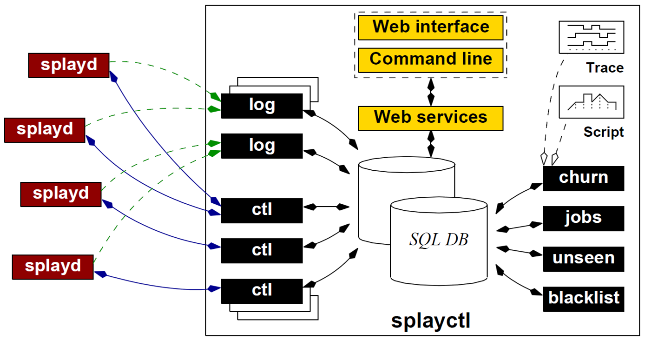

Figure 2. Architecture of the SPLAY controller `splayctl`.

SPLAY is an integrated system that facilitates the design, deployment and testing of large-scale distributed applications. SPLAY covers all aspects of the development and evaluation chain. Algorithms that run on SPLAY are written in and extended version of Lua, a high-performance scripting language. The conciseness of Lua added to the simplicity and straightforwardness of using SPLAY libraries make that algorithms written for SPLAY highly resembles the pseudo-code found in research papers. SPLAY applications are executed in a sandboxed environment that shields the host systems and enables SPLAY to also be used on non-dedicated platforms or classical testbeds like PlanetLab or ModelNet.

The SPLAY architecture comprises three main components. Figure 1 exemplifies a typical usage case. The **controller** (`splayctl`), is a trusted entity that controls the deployment and execution of applications. A lightweight **daemon** process, or `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 `splayds` daemons on participating hosts.

The controller (see Figure 2) manages applications, offering multi-criterion resource selection, deployment control, and churn management by reproducing the system's dynamics from traces or synthetic descriptions. It consists of a set of cooperating processes and executes on one or several trusted servers, sharing a single database.

The deployment of a distributed application is achieved by submitting a job locally through command-line or remotely through Web-based interface. Once registered in the database, jobs are handled by `jobs` processes, who send SPLAY applications to the daemons to be executed. Multiple SPLAY applications can run simultaneously on the same host, and the `splayctl` supports multiple users simultaneously accessing a single testbed.

3. SPLAY novel features

In this Section, we present the novel features that have been added to the SPLAY framework:

- The support for user-level network topologies emulation (Subsection 3.1);
- A novel library implementing an agreement service based on the Paxos algorithm (Subsection 3.2);
- The support for native libraries shipping, allowing to use binary libraries in the context of SPLAY jobs without the need to pre-deploy them when sending the SPLAY daemons on the testbed's nodes (Subsection 3.3);
- The support for a batch-mode operation of the controller, allowing to queue and schedule jobs in the future and based on resources availabilities (Subsection 3.4);
- The support for jobs composed of multiple files, e.g., in the case of protocol composition (Subsection 3.5);
- Finally, the support for running jobs on explicitly selected `splayds`, or on the very same `splayds` that were used for a previous deployment (Subsection 3.6).

3.1 Network topology emulation

Experiments deployed on PlanetLab [4] are hardly reproducible due to the challenging nature of the platform; in particular, the dynamism of the nodes continuously joining and leaving and the high load imposed on the (non dedicated) machines are the main culprits of this situation.

These conditions represent a challenge to evaluate distributed systems, and at the same time they are a hindrance to the reproducibility of the experiments. Nevertheless, reproducing experiments is a key aspect of distributed systems evaluation.

We introduce network emulation as a novel feature of SPLAY. The goal of this new module is to allow the execution, in a controlled environment such as a private cluster, of experiments under (emulated) network conditions, as those to be found in real-world platforms. The typical use-case is motivated by the ability to reproduce the network conditions of real-world testbed such as PL: end-to-end delays, bandwidth restrictions, or link reliability.

Existing approaches require either a dedicated hardware infrastructure as in Emulab [28] and a cumbersome cluster configuration, or special kernel adaptations as in ModelNet [26]. Furthermore, these systems do not allow the simultaneous deployment of different network topologies. We introduce the support for network emulation in SPLAY to overcome these limitations.

In contrast to existing solutions, SPLAY implements the described emulation layer in *user-mode*: no special kernel modifications are required on the hosting operative system. Moreover, the packet delays and bandwidth limitations are implemented by the `splayd` daemon themselves in cooperation with SPLAY network libraries,[1] in a purely decentralized manner, removing the need of powerful intermediate machines to regulate and constrain the packet flows between the machines where the execution takes place.

Experimenters can define a network topology using an abstract description. To maximize compatibility with existing sys-

[1] Based on LuaSocket: `http://luasocket.luaforge.net/`.

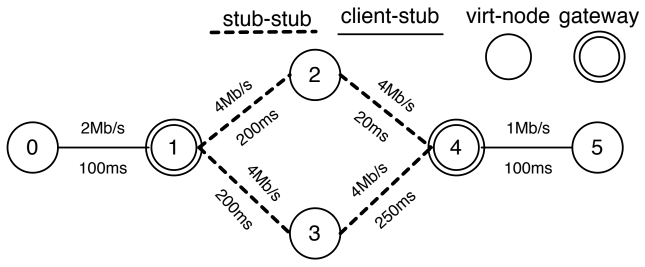

Figure 3. A graphical representation of the topology described in Listing 1.

tems, SPLAY supports the ModelNet [29] XML-based language and the Emulab TCL-based language based on the NS-2 network simulator [11]. Listing 1 gives an example of the XML topology descriptor.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<topology>
 <vertices>
 <vertex int_idx="0" role="virtnode" int_vn="1"/>
 <vertex int_idx="1" role="gateway" />
 <vertex int_idx="2" role="virtnode" int_vn="2"/>
 <vertex int_idx="3" role="virtnode" int_vn="3"/>
 <vertex int_idx="4" role="gateway" />
 <vertex int_idx="5" role="virtnode" int_vn="4"/>
 </vertices>
 <edges>
  <edge int_src="0" int_dst="1" int_idx="0"
    specs="client-stub" dbl_kbps="2048"/>
  <edge int_src="1" int_dst="2" int_idx="1"
    specs="stub-stub" int_delayms="200"/>
  <edge int_src="1" int_dst="3" int_idx="2"
    specs="stub-stub" int_delayms="200"/>
  <edge int_src="2" int_dst="4" int_idx="3"
    specs="stub-stub"/>
  <edge int_src="3" int_dst="4" int_idx="4"
    specs="stub-stub" int_delayms="250" />
  <edge int_src="4" int_dst="5" int_idx="5"
    specs="client-stub" dbl_kbps="1024"/>
 </edges>
 <specs>
  <client-stub dbl_kbps="64" int_delayms="100" />
  <stub-stub dbl_kbps="4048" int_delayms="20" />
 </specs>
</topology>
```

Listing 1. Example of network topology: 4 nodes (`virtnode`), 2 routers (`gateway`).

Figure 3 depicts the topology obtained from the XML description of listing 1. Note that each link's characteristics (delay, available bandwidth, loss rate) can override the generic values given in the section `specs`.

In a topology descriptor, two types of nodes can be used: `gateway` and `virtnode` nodes. The former represents inner nodes (*routers*) in the topology, routing packets but not supporting application nodes, while the latter represents edge-nodes and support applications' nodes.

The applications' nodes will take into account the link characteristics given by the descriptor by applying packet delays, loss rates or by limiting the packet rates (incoming or outgoing) at the sender or receiver. We assume that the physical links that exist among the `splayds` permit bandwidth rates typically orders of magnitude bigger than the emulated ones.

The descriptor is sent to the controller along with the job code. The problem of assigning a topology to the underlying physical networking infrastructure is known to be NP-hard [23]. An approximation of this solution is thus used by the `splayctl` for deciding where to deploy a given topology. Currently, the assignment of `splayds` daemon nodes to the deployed topology consists of choosing the most available nodes (highest uptime) among the less loaded ones (lowest load figure as given by the hosts' kernels). Only the `virtnode` nodes defined in the topology descriptor are objects of the assignment process. In the near future, we will consider assigning router nodes (gateway nodes) to `splayds` as required for higher accuracy in the network emulation.

Once the assigning algorithm completes, the controller resolves an *All-Pairs-Shortest-Path* over the graph induced by the topology: the weights of the end-to-end nodes are the link latencies. The resulting connectivity matrix is encoded in a compact JSON format, and it is sent to every selected `splayd` along with maximum allowed bandwidth permitted along the path to reach each of the destinations (i.e., the other `virtnodes`). Each `splayd` will therefore store a complete description of the paths between any pair of nodes. This information is exploited to emulate congestions at routers.

It is important to stress that this module allows several topologies to be deployed *concurrently*. Experiments run in isolation on top of their desired network constrictions and each node will act differently from one experiment to another. Figure 4 illustrates this feature. Two different topologies are being submitted at the same time (Figure 4-❶ and Figure 4-❸). The controller assigns topology T1 to the first three nodes (Figure 4-❷), whereas topology T2 is mapped over the complete pool of nodes (Figure 4-❹).

Implementation Bandwidth limitations are implemented using a simplified version of the *Hierarchical Token Bucket* algorithm [1]. Among other usage cases, the Linux kernel adopted this algorithm to implement bandwidth limitations [2]. A similar approach to bandwidth capping was implemented in Trickle [10]. A simple `SPLAY.topo_socket` wrapper on top of the LuaSocket module implements the bandwidth limitation mechanism. This facilitates operations inside and outside the SPLAY sandbox by exposing to the users the same APIs (e.g., for local testing outside the deployment scenario). As a result, network emulation is completely transparent to users and they do not require any code modification to use the feature. When executed within the SPLAY sandbox, we add a third wrapper after `SPLAY.restricted_socket` and `SPLAY.events_socket`. Nevertheless, we did not notice any measurable overhead on in-transit packets due to such wrapping.

Early evaluation results have confirmed the accuracy and the light weight of SPLAY user-space network emulation mechanisms. The lack of need for dedicated infrastructure or complex administration is likely to foster a broader adoption of network emulation as a tool for evaluating the characteristics of distributed systems and algorithms in a variety of conditions.

3.2 Agreement service

SPLAY already provides a set of basic mechanisms for implementing distributed algorithms: communication primitives, event-based programming, etc. We wish to extend this set of basic mechanisms with higher-level ones, that act as basic building blocks of multiple distributed applications. Indeed, reimplementing the based building blocks for each new protocol is likely to be a time-wasting and error-prone effort.

Amongst the multiple algorithms that can be deemed as inescapable building blocks are synchronization, and in particular agreement, mechanisms. We propose a new SPLAY library that supports several variants of the well-known and heavily used Paxos algorithm [18, 19].

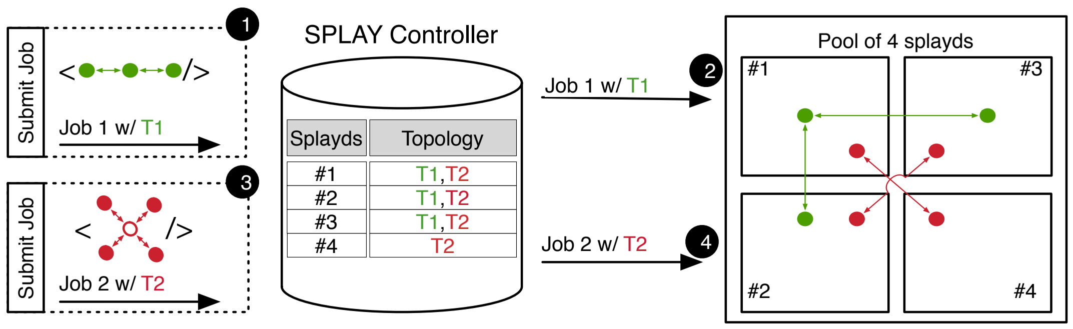

Figure 4. Submission of two jobs with network topologies: 4 `splayd` daemons concurrently run two experiments with distinct network topologies.

Leslie Lamport proposed Paxos for solving consensus between unreliable nodes. Paxos is widely used in fault-tolerant distributed systems and in particular synchronization services (e.g. Google Chubby [5], Apache Zookeeper [16]). The basic operation of Paxos is simple from the point of view of the application that leverages it. Nodes can *propose* (put) values, and query for a value. When a value is returned (get), Paxos guarantees that this is a agreed-upon value between the nodes participating to the run.

SPLAY provides now the library `SPLAY.paxos`, which contains the functions `paxos_read` and `paxos_write`. These functions allow operating a Paxos run between several nodes, and the programmer to focus on the other parts of her algorithm.

The function `paxos_read` receives the following arguments:

- An array of nodes described as `<IP address, port>`;
- A unique and consistently increasing proposal ID;
- Number of retries. When this number of retries is depleted, the function stops and returns on error.

The function `paxos_write` receives the same arguments as `paxos_read`, in addition to the `value` to be proposed.

Both functions perform RPC calls over UDP to all the nodes in the array and follow the Basic Paxos algorithm. The `read` function returns the latest stored value (which was an agreed-upon value). For the `write` function, the proposed value is stored if consensus is reached (i.e. if the majority of nodes commit to store the new value). If the proposal does not get enough quorum before a given timeout, or if any node indicates that it has already committed to a bigger proposal ID, the process is repeated. Timeouts are attributes of the library calls, and can be tuned to fit the developer's needs.

3.3 Native library shipping

We now describe an additional feature of SPLAY that allows to send binary libraries along with Lua code. Previously, only libraries deployed with the daemons on the nodes of the testbed could be used by SPLAY applications. This is desirable when one wants to ensure that only sandboxed-ready libraries are used (e.g., in non-dedicated environments) but in other conditions (e.g., testing protocols including proprietary code or code using low-level hardware acceleration) it is desirable to leverage native library, e.g., written in C. These libraries can prove more efficient than their Lua counterpart in the case of computationally-intensive operations, and the flexibility offered to users to deploy their own libraries reduces the burden on the administrator, who does not need to provision them in advance, when deploying the daemons on the testbed nodes.

To implement it, we take advantage of the existence of the `splayctl` that we use as a central repository where all submitted lib will be stored. These libraries are subsequently shipped to every `splayd` when they receive a new job. Obviously, sending binary libraries along with jobs increases the size of every deployment. We address that by adding a cache on the `splayd` so that they receive the same library from the `splayctl` only once.

Features The native library shipping features the following key aspects:

- We provide per job isolation. Every job has its view of the library set it can use, and is not able to link or load libraries from other jobs, even when deployed on the same nodes by the same controller.
- We support libraries deduplication. Native libraries sent with the SPLAY jobs are cached by each `splayd` supporting them, and exactly similar ones link to the same file, through a checksum mechanism;
- We support garbage collection, by keeping a list of available libraries at the controller, and periodically removing the instances of libraries that are deleted on the `splayctl` from the `splayd`. This is a safe operation since it is not possible for a user to deploy a job using libraries that are not registered at the `splayctl` anyhow.

In Figure 5 we present a typical scenario where an administrator and a user interact with the controller and their result on the corresponding `splayds`. The purpose of this example is to show how the set of libraries available evolves with user interactions. Even if Figure 5 labels libraries with letters as symbols, it is not the case in practice. Instead we use their real name that end with `.so` and format, which are typical `ELF` shared objects for the Linux platform [9] or `MACH-O` the shared library format for Mac OS X [3].

The first step is the submission of a job with a dependency that exists on the controller as well as on the `splayds` i.e. in their cache (Figure 5-❶). The second step is the execution of the submitted job on the `splayds` that have the library `A` deployed (Figure 5-❷).

The third step shows the submission of a library `D` for two different platforms (Figure 5-❸). It is important to state that the user is responsible for the correctness of the information sent to the controller regarding the platform, the architecture or the version of a library. No validation occurs on the `splayctl` to ensure the validity of this information. Hence, faulty information can lead to the shipment of the wrong library to a `splayd` and for example in the case of an OS mismatch a failure to execute.

The library is then available for `splayds` 2 and 4 due to their matching platforms. The matching is based on the information the `splayds` send to the controller upon registration and the information the user provided for the required library.

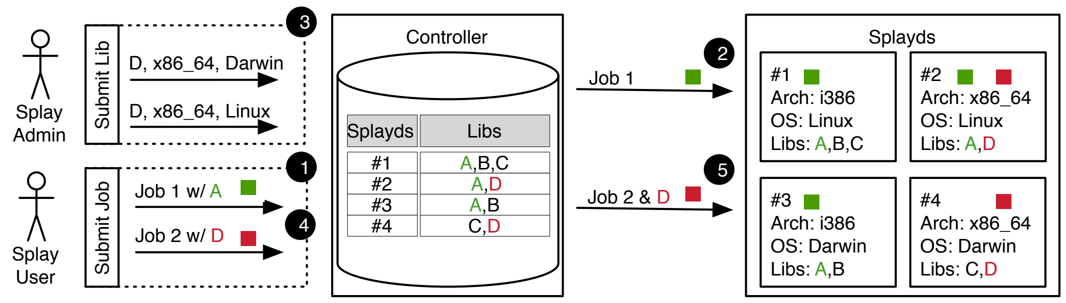

Figure 5. Native library mechanism.

The fourth step is the submission of a job with `D` as a dependency, the job is accepted provided that there are enough `splayds` that match the platform for which the library is available (Figure 5-❹).

Finally, step five show that the job is sent to the `splayds` along with required library (Figure 5-❺). The `splayd` stores the library as a file in its cache in a dedicated folder. The naming of the file changes from its original name to another one based on its `sha1` checksum. This ensures that no name conflict can occur on the `splayd` side. However, the `splayd` creates a link in the `jobd` disk space with the appropriate name. The result is that `splayds` will grow their cache of native libraries whenever they are selected for a job that requires a dependency not yet available on them.

On the controller, it is possible to avoid both the submission of libraries to the controller and the execution of jobs that declare a library as a dependency. This prevents a scenario where a controller could accept jobs requiring native libraries submitted during a previous run.

3.4 Job submission facilities: queuing, scheduling, multiple files and cloning

We refactored the submission model offered by the `splayctl`. In particular, we propose a batch-mode operation, including time-based scheduling support, resource-availability-based scheduling (queuing).

Queueing In a shared experimental environment such as a typical SPLAY cluster, it is possible that not enough resources are available for a given job at the moment of its submission.

In the original SPLAY submission model, a job was rejected if there were not enough resources to execute it (i.e. not enough nodes available, not enough `splayds` of a given version, or other selection criteria). We now define this mode of execution as *strict* and can be specified upon the submission of a job with the `--strict` flag. We introduced a new (default) job submission model for job submission to permit SPLAY users to submit a job in a controller-managed queue. A job is queued until the required resources are made available, or until a timeout expires.

Scheduling We also introduce the possibility to schedule the execution of a job at given time, given in absolute date and time or relative to the submission time. This is indicated with the flag `--absolute-time`. A SPLAY user can simply specify an execution time (and no date), to indicate that the job will be executed the same day as the submission time. This is indicated with the flag `--relative-time`.

3.5 Support for multiple file jobs

One important limitation of the previous version of SPLAY for sending complex applications or applications composed of multiple, independently implemented protocols, was that the job code was to be submitted as a single file.

We extended the default submission model, allowing the submission of jobs constituted by several Lua files. An in-house merging algorithm statically analyzes dependencies among files to define an appropriate merging strategy. In particular, our merging operation support the presence of multiple starting threads, and merge them (unless impossible) to a single control thread for the composition. Renaming and scoping allows preventing any risk of name collision.

This feature is particularly useful in the context of distributed development teams to work on different aspects of a protocol that are logically organized across distinct files.

3.6 Explicit `splayd` designation & job cloning

As an additional enhancement to the default submission model, we implemented support for running jobs on designated `splayds`. There are two mechanisms that support the new feature. As a first option, the `-splayds` flag allows the user to specify the `splayds` on which a job will be run, upon submission of a job. Alternatively, the `-splayds-as-job` flag can be used to run a new job on the same `splayds` used by a previous job. This feature is particular useful on testbeds composed on heterogeneous nodes (e.g., PlanetLab), where algorithms must be compared under the very same conditions, and hence on the same nodes.

4. SPLAY improvements

In this Section, we present the major improvements we have bring to the platform, including:

- The support for more scalable and efficient churn management through decentralized control (Subsection 4.1);
- New set of libraries for advanced operations required for distributed algorithms design, such as cryptography operations, new serialization mechanisms and more (Subsection 4.2);
- Improvements to the *command line* interface to the controller (Subsection 4.3);
- Improvements to the *web-based* interface (SPLAYWEB) for `splayds` monitoring (Subsection 4.4)

4.1 Decentralized churn management

SPLAY's churn management facilities allow to replay a trace of `ON/OFF` actions to the nodes of an application deployed on a testbed. Our original design for this operation was centralized: the `splayctl` sent on/off actions directly to the `splayds` supporting the nodes, instructing them to instantiate or kill a node, respectively. It appeared that with complex traces (with a high level of churn, that is, multiple such on and off commands to process per second),

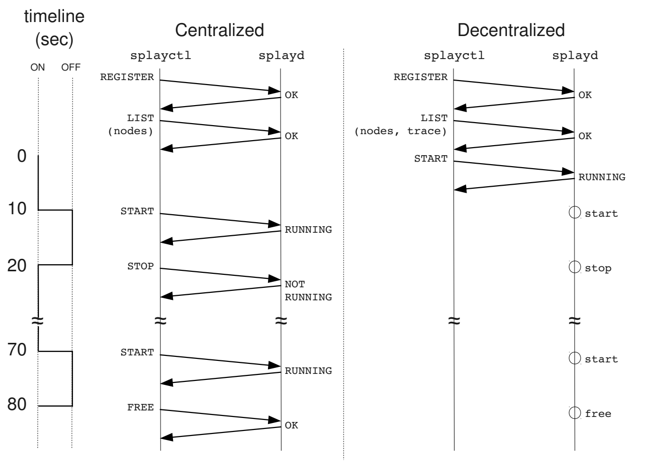

Figure 6. Centralized vs. Decentralized Churn Management: Message flow between the `splayctl` and a `splayd`.

the management of the churn replay on the controller could form a bottleneck.

We therefore replaced the centralized churn replay management by a fully decentralized one, requiring little action from the `splayctl` during the churn trace replay. This alternative decentralized churn management is performed on each of the `splayds`.

In the novel decentralized churn management, the initial `LIST` message contains not only the list of nodes as previously but also the complete churn trace for this particular `splayd`. Instead of sending `START` and `STOP` messages to the `splayds` from the `splayctl`, the controller sends at time 0 a `START` message to all nodes involved in the churn trace (see Figure 6). The process `splayd.lua`, part of the `splayd` runtime, triggers a periodic call of a coroutine that triggers the start or stop of `jobd` (components of the runtime supporting an application's node) according to the churn trace provided by the controller inside the initial `LIST` message.

Decentralized replay of the churn trace provides the following advantages:

- Processing load is reduced on the `splayctl` and supported by the `splayds`. This improves the scalability.
- Less bandwidth is used for the communication between the controller and the `splayds`. A sequence composed by several `START` and `STOP` involves only one message where all these timestamps are sent together, then `splayds` takes care of them.
- Since the controller is no longer a potential bottleneck when managing complex churn traces, the execution of the trace is more reliable, even when handling complex traces for hundreds of nodes.

We performed an evaluation of the reliability of churn replay on a testbed of six machines, five machines hosting 100 `splayds` each and the sixth one hosting the `splayctl` and database server. We used a simple churn model: eight `START` and `STOP` commands per `splayd`, with a period of 10 seconds. This represents a heavy churn scenario. For the case of 40 nodes, for example, there are four churn operations every second (all of the 40 nodes are subject of a churn operation within a time frame of 10 seconds). We tested submissions with 20, 40 and 80 nodes. An example of these churn traces is given in Listing 2. In this specific example, the trace says that the third node must be turned ON at 2 seconds from the start, then OFF at 12 seconds, ON again at 22 seconds, and so on.

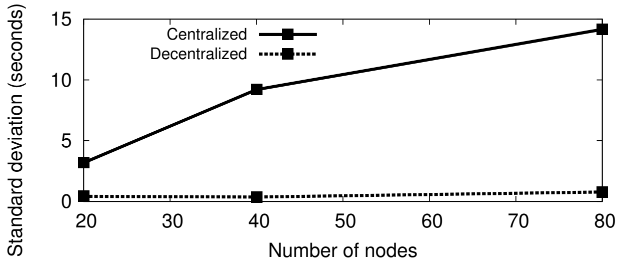

Figure 7. Standard deviation of the start/stop times with centralized and decentralized churn management.

```
0  10  20  30  40  50  60  70
1  11  21  31  41  51  61  71
2  12  22  32  42  52  62  72
3  13  23  33  43  53  63  73
...
19  29  39  49  59  69  79  89
```

Listing 2. Example of churn trace, 20 nodes.

Figure 7 presents the evaluation results. The x axis represents the number of nodes used for the submission. The y axis represents the standard deviation of the effective START and STOP times w.r.t. the times set in the churn traces. The graph shows standard deviations of the requested time and the effective time of more than 10 seconds when using the centralized churn management, while the decentralized churn management does not surpass 1 second.

Support for Churn-Aware Bootstrapping To facilitate distributed protocol bootstrap (e.g., filling an initial view for a node in an overlay network), the `splayctl` can ship a list of available nodes upon creating a job, to all `splayds` involved, which was directly available from the Lua code in a `job.nodes` variable. In the case of churn replay, this mechanism did return the nodes that were available just before the moment of activation of the application node, but it did not reflect the activations and deactivations that happened after it.

We modified the behavior of the node list mechanism to support this feature. The variable `job.nodes` has been changed to the function `job.nodes()` that returns a table with the updated list of nodes that are currently ON. We note also that this call can be used to emulate a failure detector, which is a key component of many distributed protocols. The new `job.nodes()` emulates the ideal case where all node failures and arrivals are known immediately, with no requirement of coding from the developer's side.

4.2 Improvements to Existing Libraries and New Libraries

We present below a condensed list of the improvements to the previous libraries and novel ones that have been added to the standard SPLAY deployment package.

- **Cryptography** Common symmetric and asymmetric cryptographic operations are exposed by two new modules (SPLAY.*aes* and SPLAY.*rsa* respectively). We additionally plan to contribute higher-level libraries, e.g. to support Chaum mixes and onion-routing for privacy-preserving systems.
- **Serialization** We extended the set of message encoding mechanisms available to any SPLAY applications. We introduced a

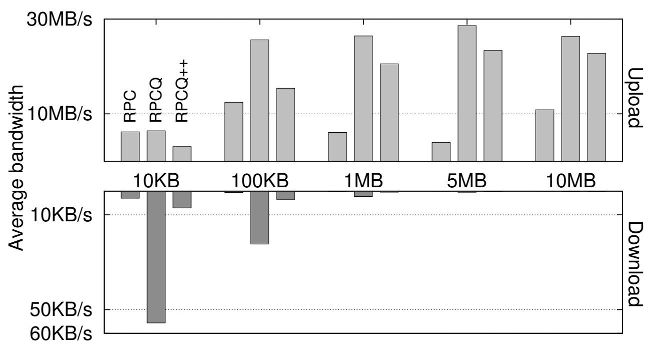

Figure 8. Bandwidth consumption for RPCQ library.

new binary-based serialization protocol, usable as drop-in replacement for the default one. This novel serialization mechanism improves on speed by orders of magnitude thanks to its underlying native library.

- **Efficiency of RPC over TCP-pools** We greatly improved the bandwidth consumption requirements of the SPLAY.`rpcq` module (remote procedure calls over TCP connection maintained in a pool with a LRU replacement strategy). The improvement is particularly noticeable for short-lived transfers of small data chunks. Figure 8 illustrates the improvements for data chunks of increasing sizes and a transfer time of 10 seconds between a simple client-server architecture.

4.3 Command-line interface improvements

Nowadays, services on Internet are often provided through Web-service APIs. This opens the possibility to easily deploy clients of such services just by developing them on top of HTTP clients. We refactored the SPLAY interface to its command line to follow this principle. The adaptation of the SPLAYWEB will follow.

The original design of SPLAY included a basic set of commands that can only be invoked locally on the machine that contains the SPLAY database, as these command line instructions directly interfaced with the DB. These commands allowed SPLAY users to submit and kill jobs, list the SPLAY daemons currently registered on the SPLAY DB, and retrieve the logs of their previous or current runs.

With the presence of a Web-service Interface, SPLAY users can now send commands from remote locations and more importantly, without an access to the database server. We expose these APIs as Web-Services using the JSON-RPC protocol, via a simple, specific HTTP server implemented in Ruby. This allows third parties to develop and propose new interfaces to SPLAY independently of the ones we already provide (Web-based, command line).

The new APIs to the controller include a more comprehensive set of commands:

- **User management** We make use of the users table implemented for SPLAYWEB (and now implemented by default) to support differentiated user types, regular and administrators. Administrators can list, add or remove users, and any user can change her password from the command line without administrator action as previously.
- **Session management** In order to manage `jobs` or `splayds`, a user must start a session, identified by a unique key, that lasts for 24 hours. Any further commands are authenticated only through the session ID.
- `Job management jobs` can be listed, submitted and killed. A SPLAY user can also retrieve logs or details about any job that she previously submitted, or any job if the user is an administrator. The command `get-job-details` returns the name and description of the job (if any), the daemons that executed the job and the job current status (e.g. "KILLED", "ENDED", "RUNNING")

We included a remote command-line interface, based on Lua scripts, that matches all the JSON-RPC calls of the Web-service interface.

4.4 `splayds` Monitoring within SPLAYWEB

Finally, we enhanced the SPLAYWEB web-based interface with a monitoring infrastructure that allows the visualization of `splayds`-profiling data in real-time. Our monitoring mechanism displays information regarding the `splayd` load and the system load for each machine that hosts a `splayd`: number of jobs running, CPU load, bandwidth consumption, memory usage and disk load. Furthermore, our monitoring infrastructure permits the visualization of the profiled data both individually, for each `splayd`, and in an aggregated form, for all the registered `splayds`.

5. Future Work

In this section we detail the direction of our future work on the SPLAY platform.

Network topology emulation We will extend the topology descriptor parser to allow even richer topologies; in particular, we will include the support to specify nodes behind Network Address Translator (NAT) devices by leveraging a module developed in our previous work [24]. We plan to further extend this mechanism to have `gateway` nodes play an active role and ameliorate the balance of (concurrent) flowing packets. Finally, we would like to take into account ongoing experiments and their underlying topologies to further improve the assignment process in terms of load balancing and resource utilization.

Agreement service The function `paxos_operation` will allow in the future to perform other Paxos modes, such as Fast Paxos [20] and Multi-Paxos [7].

Filesystem population We will provide means for an user deploying a SPLAY application to pre-populate the virtual (sandboxed) file-system of the `splayds` onto which her application will run. Upon job submission, she will be able to provide an archive, which is transferred to the selected `splayds` and then decompressed and copied into the job's sandboxed filesystem for each of the daemons. The infrastructure needed to transfer the compressed file to the daemons can be built from the code used for the submission of native libraries.

Scheduling It is sometimes better to run a job in isolation over a cluster to reduce the risk of saturating the hardware resources available to the `splayds`. We plan to improve the support for scheduled jobs to take into account the requirement of exclusive access to the test bed. The scheduling mechanism will be extended to allow submission of chain-of-jobs: it will be possible to submit a job so that its execution is scheduled upon the termination of a previously executed/currently scheduled job (i.e. execute job B as soon as job A finishes or it is killed).

Native libraries As of now, every user can upload a native library for everyone to use. This can be considered unsafe and we plan to address this issue. A first option that requires very little work is to restrict this right to administrators. A second option, and the one we will favor, is the creation of a group whose members would be allowed to upload libraries in the system.

Simulation support with SimGrid Finally, we plan to integrate SPLAY with the SimGrid [6] simulation framework. While the primary goal of SPLAY is, and shall remain, the support of live deployments on real testbeds (with or without the use of resource emulation), support for simulation would present multiple advantages. First, it would allow testing the impact of underlying layers for the distributed SPLAY applications, allowing to finely understand the influence of some distributed algorithm with, e.g., a network congestion management mechanism. Second, it will be a good starting point for proposing a debugging facility for SPLAY, with deterministic replay of events, complete event logging, etc.

6. Conclusion

The goal of SPLAY is to simplify the development, deployment and evaluation of large-scale distributed applications. In this technical report, companion to our original publication [21], we described a set of contributions that improve further the overall value of SPLAY, by making it more efficient and robust and by providing new tools that facilitate even more the job of developers.

The planned future work on SPLAY will continue towards this direction, with new features like filesystem pre-population or the enhancement of current features like native library shipping, synchronization service or network topology emulation.

This work was supported by Swiss National Foundation under agreement numbers 102819 & 200021-127271/1. SPLAY is publicly available at `http://www.splay-project.org`.

References

[1] HTB for Linux. `http://luxik.cdi.cz/~devik/qos/htb/`. [Online; accessed 25-January-2012].

[2] Linux Advanced Routing and Traffic Control. `http://lartc.org/`. [Online; accessed 25-January-2012].

[3] Mac OS X ABI Mach-O File Format Reference. `http://developer.apple.com/library/mac/#documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html`. [Online; accessed 30-January-2012].

[4] URL `http://www.planet-lab.org`.

[5] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI)*, pages 335–350. USENIX Association, 2006.

[6] H. Casanova, A. Legrand, and M. Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, Mar. 2008.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-616-5. doi: http://doi.acm.org/10.1145/1281100.1281103. URL `http://doi.acm.org/10.1145/1281100.1281103`.

[8] B. Cohen. Incentives build robustness in bittorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, jun 2003.

[9] T. Committee et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee*, 1995.

[10] M. Eriksen. Trickle: A userland bandwidth shaper for unix-like systems. In *Proc. of the USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005.

[11] K. Fall. Network emulation in the vint/ns simulator. In *Computers and Communications, 1999. Proceedings. IEEE International Symposium on*, pages 244–250. IEEE, 1999.

[12] P. Felber, P. Kropf, L. Leonini, T. Luu, M. Rajman, and E. Rivière. Collaborative ranking and profiling: Exploiting the wisdom of crowds in tailored web search. In *Proceedings of DAIS'10: 10th IFIP international conference on Distributed Applications and Interoperable Systems*, Amsterdam, The Netherlands, jun 2010.

[13] P. Felber, L. Leonini, T. Luu, M. Rajman, E. Rivière, V. Schiavoni, and J. Valerio. Cofeed: privacy-preserving web search recommendation based on collaborative aggregation of interest feedback. *Software: Practice and Experience (accepted for publication)*, 2010.

[14] P. Felber, M. Rajman, E. Rivière, V. Schiavoni, and J. Valerio. Spads: Publisher anonymization for dht storage. In *Proc. of the 10th IEEE International Conference on Peer-to-Peer Computing (IEEE P2P'10)*, Delft, The Netherlands, aug 2010.

[15] P. Felber, A.-M. Kermarrec, L. Leonini, E. Rivière, and S. Voulgaris. Pulp: an adaptive gossip-based dissemination protocol for multi-source message streams. *Springer Peer-to-Peer: Networking and Applications (in print)*, 2011.

[16] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, pages 11–11. USENIX Association, 2010.

[17] R. Ierusalimschy, L. de Figueiredo, and W. Celes. The implementation of lua 5.0. *J. of Univ. Comp. Sc.*, 11(7):1159–1176, 2005.

[18] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[19] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):51–58, dec 2001.

[20] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[21] L. Leonini, E. Rivière, and P. Felber. SPLAY: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI'09: Proceedings of the 6th Symposium on Networked Systems Design and Implementation*, pages 185–198. USENIX, apr 2009.

[22] M. Matos, V. Schiavoni, P. Felber, R. Oliveira, and E. Riviere. Brisa: Combining efficiency and reliability in epidemic data dissemination. In *Proc. of IPDPS'12: 26th IEEE International Parallel and Distributed Processing Symposium (best paper award)*, Shanghai, China, may 2012.

[23] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communication Review*, 33(2):65–81, 2003. ISSN 0146-4833.

[24] V. Schiavoni, E. Rivière, and P. Felber. Whisper: Middleware for confidential communication in large-scale networks. In *Proc. of ICDCS'11: 31st Int'l Conference on Distributed Computing Systems*, Minneapolis, Minnesota, USA, 2011.

[25] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, feb 2003.

[26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI'02*, pages 271–284, 2002.

[27] J. Valerio, P. Felber, M. Rajman, and E. Rivière. Cada: Collaborative auditing for distributed aggregation. In *Proc. of EDCC'12: Ninth European Dependable Computing Conference*, Sibiu, Romania, may 2012.

[28] 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. *SIGOPS Oper. Syst. Rev.*, 36(SI):255–270, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844152.

[29] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM'96. Fifteenth Annual Joint Conference of the IEEE Computer Societies. Networking the Next Generation. Proceedings IEEE*, volume 2, pages 594–602. IEEE, 1996.