Simplifying Hands-On Teaching of Distributed Algorithms with SPLAY

Etienne Rivière
Institut d'Informatique, Université de Neuchâtel, Switzerland
etienne.riviere@unine.ch

***Abstract*—Teaching distributed algorithms using a learning-by-doing approach is usually associated with a slow and cumbersome learning process for students. In order to test and evaluate even simple protocols, students need to learn how to set up and operate a testbed, and to write scripts for deploying, running their program and finally retrieving some logs for parsing and observation of the results. Moreover, the amount of code required in languages such as C# or Java for implementing even a simple protocol is often one order of magnitude more than the length of the protocol pseudo-code description as discussed in class or in research papers. Nevertheless, teaching distributed algorithms and protocols using real deployed code on real conditions is highly desirable and can not always be replaced by the use of simulators.**

We present in this paper our experience of using SPLAY, a distributed systems evaluation framework that greatly simplifies the work of both instructors and students for hands-on learning of distributed systems. SPLAY simplifies the writing, deployment and observation of distributed algorithms and the management of test environments, narrowing the complexity gap between pseudo-code descriptions and executable implementations. In addition, SPLAY's features and the focus kept on the algorithms and their evaluation, allow students to evaluate their protocols in a variety of conditions, by controlling the experiments and their running conditions, or by allowing running them on multiple testbeds at no additional costs and with minimal administration complexity.

***Keywords*-distributed systems; distributed algorithms; education; deployment; evaluation; learning-by-doing**

I. INTRODUCTION

A learning-by-doing approach to distributed algorithms. Education in distributed algorithms and protocols is undeniably one of the important components of today's computer science students education program. The shift from single-node applications to distributed systems and infrastructures (as exemplified by the current massive shift to cloud computing infrastructures and services) requires new generations of engineers and computing experts to at least understand the specific challenges associated with distributed programming and distributed systems design and evaluation.

While the very foundations of distributed systems, such as agreement protocols, synchronization, or naming, can be taught based on formal methods, reading and class discussions, many aspects of distributed systems design take a considerable benefit from being taught using hands-on implementation and principled observation by students. Examples of such protocols include, but are not limited to, routing and indexing mechanisms (e.g., distributed hash tables [1], [2]), scalable data stores (NoSQL databases [3], [4]), randomized large-scale (Gossip-based) algorithms [5]–[8], or P2P systems for data sharing [9], distributed file systems [10] or decentralized streaming [11]. In particular, while it is possible to convince students of the correctness of most protocols based on logical thinking, some aspects that are of paramount importance for distributed systems validation require experimentation and observation on moderate to large number of nodes to be correctly understood. These aspects comprise performance and availability, resilience to node population dynamics and failures (*churn*), performance under massive access patterns (*flash crowds*), influence of protocols composition [12], to name a few.

Challenges. While teaching distributed algorithms with a learning-by-doing methodology is desirable, it also imposes a considerable work load on students and instructors alike. This is not specific to the teaching of these aspects though: distributed systems researchers also face the same challenges when evaluating prototype implementations of their protocols. We present a high-level view of the workflow followed by a student for performing an assignment that require her to deploy an implementation of a protocol seen in class or in her readings, and to extract a graph or some statistics from a distributed run.

First, there is usually a significant complexity gap between the (usually concise) pseudo-code description of algorithms taught in class or presented in research papers, and the complexity of their implementation in languages widely used for computer science education such as C# or Java. This requires students to learn a new set of libraries, write housekeeping code for establishing connections, bootstrapping an overlay, setting up timers and event handlers, marshaling arguments and dealing with stubs for RPC invocation, and more. While these aspects of languages are certainly important to master by students in their whole curricula, we argue that the complexity of implementation too often hinders the fundamental goal of teaching distributed algorithms and their evaluation. Too often, due to the limitations in time of students' lab hours, only a limited number of protocols can be reasonably expected to be implemented. Another observation we made is that students tend to concentrate a larger part of their efforts on the implementation complexity rather than on the actual evaluation of the properties of the algorithms.

Second, deploying live code on a test infrastructure (which

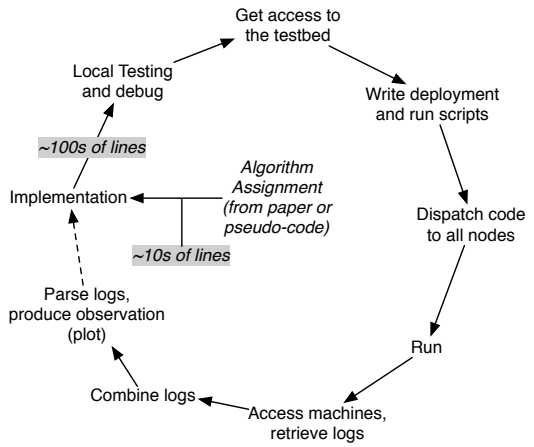

Figure 1. A typical workflow for a student working on a distributed algorithm evaluation assignment with standard methods and languages.

we shall refer to as *testbed*) is also a complex and poorly rewarding process for students, instructors and teaching assistants. A first observation is that a testbed must be configured to allow access by students. While for a local cluster dedicated to this usage, this can be done by using the university authentication system, this becomes fairly time-consuming for testbeds such as PlanetLab,[1] or for more specific testbeds such as a cluster with a Modelnet[2] node for network emulation.

Third, deploying on the testbed requires a student to write scripts that will (1) send her compiled code to all nodes of the testbed (most of the time along with the complete set of required libraries); (2) remotely run the code and redirecting outputs to files or remote connections; (3) retrieve these logs on her workstation and (4) combine them for extracting meaningful data using a tool such as R.[3] In our experience, this operation is very error-prone and, while posing no particular technical challenge, ends up consuming a fair amount of students' time, again distracting them from the core of the teaching. Moreover, these scripts are testbed-specific: asking students to test their algorithm on both a cluster and PlanetLab requires them to write new scripts.

As a result of the previously mentioned limitations, time is often too limited in one semester to really delve into the most interesting aspects of distributed algorithms evaluation, such as the behavior of algorithms under churn, the impact of nodes failures or particular network topologies, comparison of several algorithms for the same task (e.g., a comparison of DHT protocols), etc. We also stress out that support for experiment control, e.g., mimicking churn or emulating network topologies (delays, packet loss) is often limited and also requires new scripts to be written by the student or provided by the instructor.

Why not simulations? As previously said, the challenges mentioned above are also faced by distributed systems researchers, and their answer to the complexity gap between pseudo-code algorithms and running implementations is often to use simulations. While simulators can arguably be useful in some cases for teaching distributed systems, they are by no means silver bullets. Often, either a simulator is simplistic (e.g., assume complete synchronicity and no network topology) but simple to use, or it is accurate in approaching the reality of a given class of networks, but at the price of a complex and/or slow operation. We firmly believe that real deployment should be favored as they are better suited for students to face the challenges and difficulties that await them when building distributed applications in their professional life.

Outline. We started developing some time ago a framework, named SPLAY [13], that seeks to greatly simplify the development, deployment and experiment control for distributed systems evaluation. The original goal of SPLAY was to facilitate the rapid prototyping and evaluation of distributed algorithms, such as overlay networks or P2P systems, by computer science researchers.

In the remainder of this paper, we present our experience of using SPLAY in the context of a graduate course on large-scale distributed systems at the universities of Neuchâtel, Bern and Fribourg in Switzerland. The subject of this course is on the design principles and algorithms for large-scale (>1,000 nodes) systems. It focuses more on understanding the general behavior of large-scale systems building blocks rather than on learning specific technologies. As such, this course follows a research-oriented approach, where algorithms and protocols are taught based on the observation of their global properties, such as topological and structural characteristics, performance, resilience to dynamic conditions, or self-* abilities. We start by describing SPLAY in the next Section, present our experience of teaching distributed systems using SPLAY in Section III and conclude in Section IV.

II. SPLAY: SIMPLE EVALUATION OF DISTRIBUTED ALGORITHMS AND SYSTEMS

SPLAY is a distributed systems development, deployment and experiment control framework. We only give a brief introduction to SPLAY. Details and evaluation can be found in its reference paper [13]. SPLAY code is open source. It can be downloaded or tried with a demo deployment on PlanetLab at http://www.splay-project.org. A high-level view of SPLAY for the purpose of teaching is given by Figure 2.

Splay architecture. SPLAY runs a set of *daemons*, one for each node of the testbed. These lightweight processes are always running and receive instructions for instantiating nodes of the students' applications from the SPLAY *controller*.

[1] http://www.planet-lab.org/, a worldwide distributed testbed with 1,000+ nodes on 533 sites, devoted to distributed systems and networks research.

[2] http://ccis.colorado.edu/2011/04/modelnet-colorado/

[3] http://www.r-project.org/

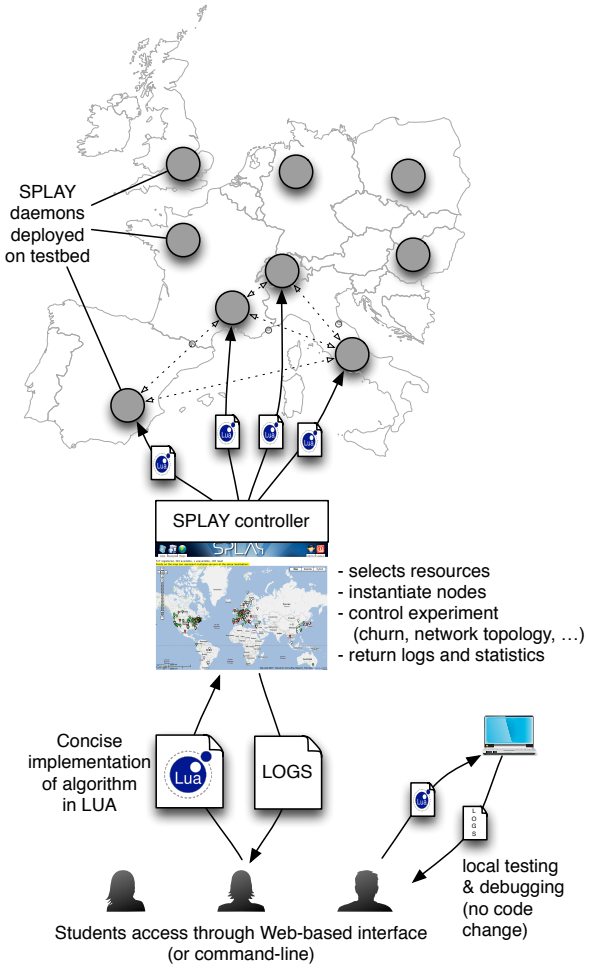

Figure 2. High-level overview of SPLAY

The controller is responsible for managing the set of daemons. For instance, it allows selecting the nodes for the deployment based on a variety of criteria: load, available memory, resource restrictions, and even geographical coordinates when applicable. It receives the application code from students by means of a simple Web interface. While a command line interface with the same functionalities is also available, we noticed that students prefer using the Web interface for multiple reasons: it allows them to browse through their previous submissions and the corresponding results, to visually inspect the state of nodes and running or terminated applications, etc. Both interfaces support multiple users and accounts, and one student does not have access to other students' submissions unless the instructor decides so.

An experiment can be controlled while it is running by having the controller send instructions to the daemons during the execution. An example is when a *churn trace* is provided along with the submission. Such a trace describes when nodes should go up, or down, according to time, and is automatically generated from a high-level description in a simple DSL. Another example is that of network topologies: SPLAY daemons are able to emulate network links according to a topology provided as a XML description, if it is the will of the instructor that students evaluate their protocols on specific network conditions.

It is important to note that this architecture removes the need of managing students' access to the testbed, as well as the complexity of using multiple testbeds as part of a course's practical sessions. Indeed, the only access is made through the Web interface, with an automatic registration process. The administrator needs to run a daemon on each node once, and students only access them *via* the controller interface.

While being conceptually a centralized entity, the controller is not a bottleneck for the typical testbeds used in research and teaching. It is itself distributed and can leverage a cluster or a multicore. With a single low-end server (4GB of RAM), there is no difficulty in operating a 1,000 nodes testbed (e.g., a PlanetLab slice) with the same amount of running daemons.

SPLAY also facilitates logging: as illustrated on Figure 2, all logs from the nodes onto which an experiment is deployed are retrieved by the student directly from the interface to the controller. These logs are simply shipped to the controller as they are generated, through pre-established links between the daemons and the controller. They can be directly processed and parsed by the student to extract meaningful information.

Splay language and libraries. The other main objective of the SPLAY design is to provide a concise, simple and efficient implementation language. The goal is to remain as close to pseudo-code as possible in terms of number of lines of code when writing distributed algorithms. The SPLAY language is an extended version of Lua,[4] a highly efficient scripting language, recognized for its smooth interaction and integration with C, and being used in a large variety of applications as a high-level integration language (examples include control systems, games, stream data processing engines, etc.). Lua's syntax is easy to learn and very close to what students are used to (sequential and procedural).

The language is supported by an extensive set of libraries that support the typical operations of distributed algorithms. Note that the complete Lua standard library is also available, which is feature-equivalent to the C `stdlib`. We extend Lua's features with support for cryptography, cooperative multitasking (which proves to be easier to use and less error prone than preemptive multitasking in this context, and also more efficient in practice), event-based programming (either local or remote, allowing message-passing-based reactive programming). We leverage the LuaSocket network library but since our goal is to reach ease-of-coding and concision, all data serialization and deserialization is made transpar-

[4] http://www.lua.org/

ently. An RPC library allows calling any function of a remote node, or even accessing (non-protected) variables. This latter feature is one of the reasons SPLAY implementations are nearly as concise as pseudo-code.

As an example, we present a code snippet with the corresponding pseudocode from the original paper describing the Chord distributed hash table [1] in Listing 1 and Algorithm 1, respectively. The complete, readily deployable SPLAY code for Chord is available in [13]. This excerpt describes the key lookup mechanism in Chord: nodes are logically organized in a ring, and data keys are mapped to the ring to nodes with the smallest greater key. Nodes are linked in a hypercube-like structure navigated through the use of a *fingers* table, until the node responsible for a key returns its identity (which is then propagated back to the initiator). We observe that the complexity of the pseudo-code and the SPLAY code is similar. Indeed, the complexity of establishing communications, serializing arguments, etc. is removed through the use of a single RPC (line 6 in both figures, the RPC call is denoted as "`call(node,{arguments})`" in Figure 1). Note also that there is no need for students to declare which function will be accessed by RPC and generate stubs for this: this is handled transparently by SPLAY libraries. In total, the Chord implementation for SPLAY directly from the pseudocode of [1] is 59 lines of code, including a main function issuing periodic insertions and requests to random keys. This is to be compared to the 44 lines of pseudocode from [1], which does not include bootstrap. While such a simple implementation is readily deployable on a cluster-type testbed, it requires a few additional modifications to be able to run under more challenging conditions, e.g., on PlanetLab. Checking for the completion (or timeout) or RPCs and returning an error code to the caller requires an additional 17 lines, while implementing a successor list and routing around failed nodes another 26 lines. In total, students are able to write a fully-functional, fault-tolerant DHT in 100 lines of code, with no particular difficulty stemming from the language or the environment, leaving them to concentrate on the algorithm evaluation.

Sandboxing and non-dedicated testbeds. The last aspect of SPLAY that has a particular importance in the context of teaching is its native support for *sandboxing*. All libraries calls, including those to the standard Lua (C-based) libraries, are intercepted with no overhead thanks to Lua's native support for function *scoping*. This allows restricting the amount of resources each node of deployed algorithms is allowed to consume (network, disk space, memory, etc.). These limits are set up by the administrator upon deployment of the daemons.

The first immediate gain is that ill-behaved code from one student (e.g., trying to consume large amount of memory, or to send at full speed over the network when it should not be necessary) will be isolated and will not (or minimally) interfere with the experiments of other students. The sandbox also allows restricting communications to a white list of IP addresses (typically, where daemons were deployed). This prevents the risk of students (accidentally or not) performing distributed denial-of-service to a public IP on the Internet, e.g., from multiple PlanetLab nodes.

```
1  function find_successor(id)              -- ask node to find id's successor
2    if between(id, n.id, finger[1].id, false, true)
           -- inclusive for second bound
3      return finger[1]
4    end
5    local n0 = closest_preceding_node(id)
6    return call(n0, {'find_successor', id})
7  end
8  function closest_preceding_node(id)              -- finger preceding id
9    for i = m, 1, -1 do
10     if finger[i] and between(finger[i].id, n.id, id, false, false) then
11       return finger[i]
12     end
13   end
14   return n
15 end
```

Listing 1. SPLAY code for implementing the Chord overlay lookup call from Algorithm 1.

Algorithm 1: Key lookup using the finger table in the Chord distributed hash table, copy of Algorithm 5 on page 5 of [1].

```
   // ask node n to find the successor of id
1  function n.find_successor(id)
2      if id ∈ (n, successor] then
3          return successor
4      else
5          n' = closest_preceding_node(id)
6          return n'.find_successor(id)
7      end
8  end
   // search the local table for the highest predecessor of id
9  function n.closest_preceding_node(id)
10     for i = m downto 1 do
11         if finger[i] ∈ (n, id) then
12             return finger[i]
13         end
14         return n
15     end
16 end
```

Second, the sandboxing feature allowed us to leverage the many machines present in students' labs at the university, to create a distributed testbed for free. Asking for a individual account for all machines for each student, and letting them run any type of code without fences, is not likely to be accepted by any system administrator. The presence of sandboxing and strong guarantees on resource consumption, along with the absence of need for individual accounts, allow to deploy SPLAY daemons on these *non-dedicated* testbeds, for teaching distributed algorithms when no cluster or Planet Lab slice can be provisioned for a course.

Workflow. Figure 3 describes the typical student workflow

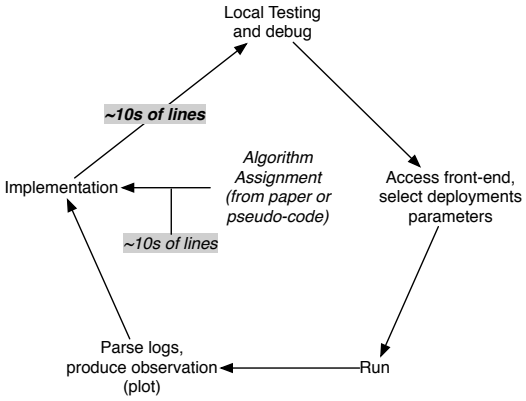

Figure 3. Workflow of operations from a student's perspective for implementing and deploying a distributed application with SPLAY

for an assignment. In contrast to the "classical" workflow of Figure 1, we can see that the student can concentrate on the interesting parts of distributed algorithms evaluation: writing and deploying rapidly, extracting information, using concise and easy to debug (and to grade) implementations.

III. LEVERAGING SPLAY IN A LARGE-SCALE DISTRIBUTED SYSTEMS COURSE

We now describe our experience of using SPLAY for the practical sessions of our large-scale distributed systems graduate course. Note that a similar (reduced) course could very well be taken by undergraduates provided they have some knowledge on networking and imperative programming.

This course covers multiple topics pertaining to large-scale distributed systems design and implementation, including (1) randomized and gossip-based algorithms, (2) self-organizing overlays, (3) distributed indexing (DHTs, but also indexing mechanisms supporting complex queries such as ranges), (4) peer-to-peer file systems, (5) MapReduce paradigm and implementation, (6) application-level multicasting, (7) file sharing applications, (8) topic- and content-based peer-to-peer publish and subscribe.

A set of eleven time slots, of two hours each, are dedicated to hands-on practical sessions. We have observed that a single of these slots is necessary to introduce SPLAY and a simple token-ring mutual exclusion example to students. In two hours, they are able to run their first distributed algorithm, write a simple parsing script and produce results using the `gnuplot` plotter, much like the typical workflow of an experienced distributed systems researcher or PhD student. Note that our student base is highly heterogeneous: roughly half of the students come from the three universities of Neuchâtel, Bern and Fribourg, the other half being international students. The students have a

The students have a command of programming that ranges from fair but restricted to specific environments (e.g., C# and .NET using MS Visual Studio) to moderate as some students come from a more theoretical background, e.g., maths-oriented undergraduate studies.

A typical assignment plan for the ten remaining weeks is as follows. (1) Students first implement the gossip-based *peer sampling service* [5], which creates random overlays and allows decentralized node membership management. Their assignment and evaluation focus on structural aspects of created overlays: not only the students validate that the behavior is correct, but they also compare system-wide criteria such as connectivity, clustering, and reaction speed to node failures/arrivals. (2) Students use this peer sampling to implement aggregation protocols as described in [6], with an application to decentralized network size estimation, and replica reconciliation and dissemination protocols described by [14]. (3) Students then implement the Chord DHT [1] that we previously described, with additional fault tolerance support, and evaluate its behavior under various churn conditions. These three assignment use the first six slots of the ten, while the remaining four slots allow them to conduct a broader study of an algorithm described in a research paper of their choice. Examples of such from previous years are the implementation and evaluation of a self-stabilizing version of Chord [1], T-Chord [8] that leverages the T-Man [7] gossip protocol, or the Scribe [15] publish and subscribe system on top of the Pastry DHT [2].

In all assignments, students have to reproduce experimentally some of the plots from the original papers or lecture notes (or to compare to simulations). Success criteria are based on reproducing and analyzing system-wide information as plots and statistics—much like researchers do.

Our general observation, based on comparison with our previous teaching experiences of distributed algorithms in other universities, is threefold. First, the number of protocols that the students can implement and evaluate within these ten weeks of practical sessions is perceptibly greater than what would have been possible using, e.g., Java and "manual" experiment management. Obviously, SPLAY does not remove the complexity of distributed programming, such as the presence of live/deadlocks, race conditions, overlay inconsistencies, etc. but these are easier to isolate and reason upon with a concise implementation. Second, the evaluation of each protocol, not being hindered by the complexity of the environment of administration issues, can be bolder for the same amount of time spent by the same student, than what she would have produced with previous approaches. Third, the use of the concise SPLAY language and its deployment facilities allows instructors and teaching assistants helping with the practical sessions to focus on the core of the course content when directing students: distributed algorithms and their evaluation, rather than loosing time helping with deployment or scripting issues, and the course to be of greater benefit to students overall according to its initial objectives of focusing on distributed algorithms evaluation.

IV. Conclusion

We have presented the use of SPLAY, a distributed algorithms development, deployment and evaluation framework initially created for distributed systems researchers, in the context of teaching large-scale distributed algorithms and protocols and their evaluation. Our experience so far has been fairly positive: the simplicity offered for implementing, deploying and observing distributed algorithms allowed the course and its practical sessions to touch upon a larger variety of distributed algorithms, and to introduce students to the principles of distributed systems evaluation and comparison.

The SPLAY platform is constantly evolving, and part of its evolution is now driven by its use for teaching. Our future evolution for the tool in this direction are as follows.

First, we would like to enhance the logging mechanism in order to support causally-ordered logs for applications based on message-passing. While it is important for students to understand there is no such thing as a global time-clock in a distributed system, this would also allow in some case easier debugging and understanding of students' implementations.

Second, we would like to propose a novel interface to the controller inside an integrated IDE that students are usually mastering, such as Eclipse. This would allow sending a distributed application directly from Eclipse, and to support additional pre-deployment debugging facilities based on the Lua debugging tools and updated SPLAY libraries.

Finally, we plan to provide under an open license a set of practical sessions assignments for use by other instructors; in order to bootstrap a common base of exercises for training future distributed systems practitioners and researchers into the highly interesting–but also highly challenging–task of evaluating distributed algorithms and systems.

Acknowledgments

The author is grateful to his colleagues from the Université de Neuchâtel who contributed to the SPLAY platform: Lucas Charles, Pascal Felber, Raluca Halalai, Lorenzo Leonini, Valerio Schiavoni and José Valerio. The Large-Scale Distributed Systems course benefited from an *innovative teaching* grant from the Université de Neuchâtel, and from the dedication of teaching assistants Dilek Harmanci, Valerio Schiavoni and José Valerio. The research leading to SPLAY received funding from the Swiss National Foundation (grants 102819 & 200021-127271/1).

References

[1] 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*, vol. 11, no. 1, pp. 17–32, feb 2003.

[2] A. Rowstron and P. Druschel, "Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, 2010.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.

[5] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Transactions on Computer Systems*, vol. 25, no. 3, aug 2007.

[6] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Transactions on Computer Systems*, vol. 23, no. 3, pp. 219–252, 2005.

[7] ——, "T-man: Gossip-based fast overlay topology construction," *Computer Networks*, vol. 53, no. 13, pp. 2321–2339, aug 2009.

[8] A. Montresor, M. Jelasity, and O. Babaoglu, "Chord on demand," in *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 87–94.

[9] 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.

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 2001, pp. 202–215.

[11] T. Locher, R. Meier, S. Schmid, and R. Wattenhofer, "Push-to-pull peer-to-peer live streaming," in *Proceedings of DISC 2007: 21st Internation Symposium on Distributed Computing*, Lemosos, Cyprus, sep 2007.

[12] É. Rivière, R. Baldoni, H. Li, and J. Pereira, "Compositional gossip: a conceptual architecture for designing gossip-based applications," *ACM SIGOPS Operating Systems Review, special issue on Gossip-based Networking*, Oct. 2007.

[13] 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*. USENIX, apr 2009, pp. 185–198.

[14] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 1987, pp. 1–12.

[15] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "Scribe: A large-scale and decentralized application-level multicast infrastructure," *IEEE Journal on Selected Areas in Communication (JSAC)*, vol. 20, no. 8, Oct. 2002.