p2psim: a simulator for peer-to-peer protocols

Thomer M. Gil, M. Frans Kaashoek, Jinyang Li, Robert Morris, Jeremy Stribling
Computer Science and AI Laboratory, MIT, Cambridge, MA
{thomer,kaashoek,jinyang,rtm,strib}@csail.mit.edu

Abstract

This paper describes p2psim, a simulator intended to compare, evaluate, and explore peer-to-peer protocols. Our primary goals are to make understanding peer-to-peer protocol source code easy, to make comparing different protocols convenient, and to have reasonable performance. Because p2psim uses threads, implementations look like algorithm pseudo-code, which makes them easy to comprehend. p2psim supports several peer-to-peer protocols, making comparisons between different protocols convenient. The design of p2psim's control flow maximizes concurrency for performance, minimizes the need for synchronization, and avoids deadlocks. p2psim supports a network delay model constructed from actual measurements of Internet hosts, as in King [2]. p2psim only models network latency, not queueing delay, bandwidth, or packet loss; this suffices for experiments where latency is the bottleneck. Peer-to-peer protocol code written for p2psim has significantly fewer lines than the real implementations and is easier to understand. We hope to encourage the peer-to-peer research community to use p2psim, a convenient tool to experiment with new ideas that could benefit future research into peer-to-peer protocols.

1 Introduction

This paper describes p2psim, a discrete event simulator intended to compare, evaluate, and explore peer-to-peer protocols. p2psim's primary goals are to make reading and writing peer-to-peer protocol source code easy, comparing different protocols convenient, while still performing reasonably. Existing peer-to-peer protocols are often written in event-driven callback style. For example, Tapestry [12] is implemented using SEDA [11] and Chord [10] is implemented using libasync [6]. Unfortunately, this programming style chops up the source code, making it hard to understand. By using threads, p2psim allows a user to write programs like pseudo-code, which makes them easier to comprehend than the real implementation. p2psim's design minimizes the need for synchronization, maximizes concurrency for performance, and avoids deadlocks. p2psim already supports Chord, Tapestry, Koorde [4], Kademlia [5], and Kelips [3]. This support for different peer-to-peer protocols makes comparing a new or modified peer-to-peer protocol against others convenient.

p2psim's network delay models only model latency, not bandwidth, queueing delays, or packet loss. This suffices for most experiments where network latency is the bottleneck. A network delay model that p2psim supports is based on a latency model constructed from actual measurements of Internet hosts, as in King [2].

p2psim abstracts away many intricacies of network protocols, leaving only the core peer-to-peer algorithms to think about. Furthermore, a user can simplify many aspects of a protocol's implementation. For example, a node can be given complete knowledge about other nodes, which makes implementing a protocol's routing table initialization straightforward.

To perform a fair comparison between different peer-to-peer protocols, they should be exercised under similar conditions. p2psim allows a user to run different peer-to-peer protocols on a specific network topology with a specific workload. Because p2psim is a simulator, a user has more control over network conditions in these experiments than in a real-life testbed. This allows a user to study the effects of certain—possibly uncommon—network conditions on different peer-to-peer protocols. Finally, scaling experiments to a large number of nodes is easier in p2psim than in real-life.

We found that implementing different peer-to-peer protocols in p2psim is fairly straightforward. Some careful coding is required to handle concurrent threads operating on one data structure, although in our experience this did not require any locking. Our measurements show that a running simulation takes 7MB of memory plus roughly 42KB per simulated node, but this strongly depends on the memory usage of the particular peer-to-peer protocol that each node runs.

We hope to encourage the peer-to-peer research community to use p2psim as a tool to experiment with new ideas. We believe that this would benefit future research into peer-to-peer protocols.

2 Related work

ns-2 is a network simulator intended for network research[1]. Like p2psim, users can write their own code and run it on various network topologies. However, ns-2 does not support threads. p2psim's use of threads allows a user to write pseudo-code style programs, keeping them simple.

SimPastry[2] is a simulator specifically built for Pastry [9]. Similarly, Chord [10] has a simulator, but it only simulates Chord. In addition, there is a simulator written specifically for SEDA-based programs, as first described by Rhea, et al. [7]. In contrast, p2psim supports several different peer-to-peer protocols.

MACEDON [8] generates peer-to-peer protocol source code for different environments based on a finite-state machine specification in a specialized grammar. We believe that MACEDON looks promising, but at the time of this writing we were unable to obtain MACEDON's source code, a peer-to-peer protocol specification in MACEDON's grammar, or source code generated by MACEDON, and thus are unable to provide an accurate comparison.

3 Design

This section describes the main components in a simulation, explains how they communicate, and discusses p2psim's control flow design.

3.1 Main components

A simulation includes Nodes, a Network, a Topology, a Lookup Generator, and a Churn Generator. Figure 1 gives an overview of these components, and how they interact.

A Node simulates a computer running a peer-to-peer algorithm. Nodes (labelled N1 to N7 in Figure 1) communicate to each other by sending packets through the Network. The Network uses the Topology to determine

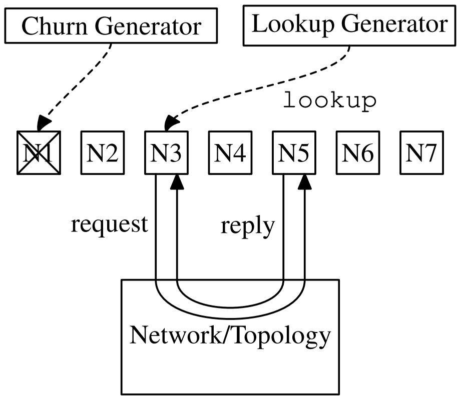

Figure 1: Major components in p2psim.

the latency between two nodes. The Churn Generator models the dynamic nature of peer-to-peer networks. Periodically, it either adds a node to the network by invoking the node's `join` method, or removes a node from the network by destroying it. Figure 1 shows the Churn Generator removing node N1. The Lookup Generator simulates a workload by periodically issuing `lookup` requests. Even though the Lookup Generator and the Churn Generator are in principle different, they are typically implemented as one component for simplicity. Figure 1 shows the Lookup Generator generating a `lookup` request for node N3. To satisfy this request, node N3 sends a packet over the Network to Node N5, N5 sends back a reply, and N3 logs the time it took for the `lookup` to complete, for later analysis by the user.

During one simulation, all Nodes run the same peer-to-peer protocol. For each new protocol, the user must implement a `join` and `lookup` method. `join` initializes a node's routing table and contacts one or more nodes to establish itself as member of the peer-to-peer network. Depending on the protocol, a node doing a `join` may contact a specific, well-known node to bootstrap itself. A node doing a `lookup` runs the protocol's lookup algorithm to find node N closest to a given key K in identifier space. Periodically, a Node may run a stabilization routine to refresh the entries in its routing table.

Nodes communicate to each other by issuing RPCs that are sent in packets through the Network. Upon receiving a packet, the Network asks the Topology for the latency between the sender and receiver, which determines the correct simulated time at which the packet

[1] http://www.isi.edu/nsnam/ns/

[2] http://research.microsoft.com/~antr/pastry/instructions.htm

is delivered to the destination Node. When an RPC is issued to a dead node, the Network mimics a timeout by scheduling a special error-RPC-reply n round-trip times after the RPC was sent, where n is some user-configurable value. The Network does not model queueing delay or bandwidth. This simple network model suffices for experiments where network latency is the bottleneck.

The Topology models the network delay; it knows the latency between any two nodes. p2psim supports several different Topologies, including one that is based on a latency model constructed from actual measurements of Internet hosts. We obtained this data set by adapting the techniques of the King [2] project to measure the round-trip times between 1900 domain name servers. The King data set provides a realistic set of latencies and captures relationships among the latencies between multiple nodes since it directly measures real Internet hosts. The King measurement technique relies on timing recursive DNS queries. To determine the distance between name servers A and B, which serve domains D_a and D_b respectively, we ask A to resolve a name in domain D_b and time this operation. By subtracting the measured latency to A from this time we obtain the latency between A and B.

3.2 Threads

p2psim uses threads to support concurrent operations for performance, and to keep the source code easy to understand. p2psim's control flow design goals are to minimize the need for synchronization, while avoiding deadlock. This rest of this section describes how we achieve these goals.

p2psim's main thread repeatedly removes the first event (e.g., a packet-delivery event) from an event queue and executes that event within a *new* thread. The events waiting in the event queue are sorted by timestamp. The current time in the simulator is the timestamp of the event removed from the event queue most recently. After the main loop creates a new thread for an event, it yields the CPU. Only when no other threads are runnable will the main thread execute the next event in the event queue.

The Lookup Generator, the Churn Generator, the Network, and Nodes can each insert events into the event queue. The Lookup Generator inserts `lookup` events into the event queue, and the Churn Generator inserts `join` events and node crashes into the event queue.

When the main thread removes a `lookup` or `join` event for a node, N, from the event queue, it creates a new thread, T, that executes the node N's handler. When thread T—now running within the context of node N—issues an RPC, it hands a packet to the Network and then *blocks*, waiting for the RPC reply. (p2psim also supports asynchronous RPCs.) The Network asks the Topology for the latency and schedules a packet-delivery event by inserting it into the event queue. When the main event loop removes this packet-delivery event, it spawns a new thread to execute the RPC handler, similar to `lookup` and `join` events. When the RPC reply eventually comes back, thread T unblocks. A thread exits when the handler it executes returns. A Node can also schedule a callback to itself; this is necessary, for example, to periodically invoke a stabilization routine.

Because each event runs in its own thread, we avoid deadlock and increase concurrency. Because p2psim uses a cooperative threads package, the context switches happen at known locations in the code, for example when a thread blocks on an RPC. We found that careful coding eliminates the need for locks to do synchronization.

4 Experience

Peer-to-peer protocol code written for p2psim has significantly fewer lines than the real implementation. For example, a fairly complete implementation of Kademlia consists of approximately 2000 lines of C++ code versus approximately 14500 lines of code for the full implementation.

```
lookup_args a;
lookup_result r;
a.key = 0x8dde280;
bool ok;
ok = doRPC(ip, &K::a_handler, &a, &r);
```

Figure 2: A node issuing an RPC.

Figure 2 is a code fragment that demonstrates how a Node running protocol K issues an RPC to a node with IP address `ip`. When `doRPC` returns and `ok` is true, then `r` will contain the reply to the lookup.

We found that writing implementations for Chord, Tapestry, Koorde, Kademlia, and Kelips for p2psim was fairly straightforward. However, we also realized that p2psim's `join/lookup` API is not a natural fit for all protocols. Chord's `lookup` algorithm finds a node N closest in identifier space to a given key K. This is the modus operandi of `lookup` we had in mind when defining the API. However, Kademlia has two different `lookup` methods that don't easily translate to Chord's notion of `lookup`. Kelips has a similar problem. The lack of a unified API for different peer-to-peer protocols sometimes made comparisons between different protocols tricky. We believe that defining a common API [1] for different peer-to-peer protocols is an interesting topic for future work.

We also dealt with some programming subtleties. In particular, a thread, T1, that issues RPCs in a loop that iterates over a data structure requires careful coding. While thread T1 is blocked on the RPC, thread T2 may change the contents of the data structure that thread T1 is looping over. In general, the user should be aware that a node's state may change across RPCs.

A simulation with 1024 nodes running a protocol that stores no state takes up approximately 7MB of memory. The same simulation with 1024 nodes running Kademlia takes up approximately 51MB of memory, which amounts to roughly 42KB of data per node. Of course, the exact memory requirements depend largely on the memory usage of the simulated peer-to-peer protocol.

5 Conclusion

This paper describes p2psim, a simulator intended to compare, evaluate, and explore peer-to-peer protocols. p2psim makes understanding peer-to-peer protocol source code easy, makes comparing different protocols convenient, and has reasonable performance. Because p2psim uses threads, implementations look like algorithm pseudo-code, which makes them easy to comprehend. p2psim supports several peer-to-peer protocols, making comparisons between different protocols convenient. One of p2psim's network delay model is constructed from actual measurements of Internet hosts. We hope to encourage the peer-to-peer research community to use p2psim, because it is a convenient tool to experiment with new ideas that could benefit future research into peer-to-peer protocols.

References

[1] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common api for structured peer-to-peer overlays, 2003.

[2] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proceedings of the SIGCOMM Internet Measurement Workshop (IMW 2002)*, Marseille, France, November 2002.

[3] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03)*, 2003.

[4] M. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table.

[5] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric, 2002.

[6] D. Mazières. A toolkit for user-level file systems, 2001.

[7] S. C. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proceedings of INFOCOM 2002*, 2002.

[8] A. Rodrigues, C. Killian, S. Bhat, D. Kostić, and A. Vahdat. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. Technical report, 2003.

[9] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[10] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, Aug. 2001. An extended version appears in ACM/IEEE Trans. on Networking.

[11] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.

[12] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE Journal on Selected Areas in Communications*, 2003. Special Issue on Service Overlay Networks, to appear.