Brief Announcement: Building Data Structures on Untrusted Peer-to-Peer Storage with Per-participant Logs

Benjie Chen, Thomer M. Gil, Athicha Muthitacharoen, and Robert Morris
{*benjie,thomer,athicha,rtm*}*@lcs.mit.edu*
MIT Laboratory for Computer Science

Introduction

Structured peer-to-peer distributed hash tables (DHTs) provide a simple API for reading and writing key/value pairs (often called blocks). A DHT typically takes care of finding a network host to store each key/value pair; replicating data for availability; and checking that retrieved blocks have not been tampered with. The DHT interface is fairly low level, much like the sector read/write interface of a disk drive. A DHT-based distributed application typically maintains complex data structures on top of the DHT, with blocks containing pointers (keys) to other blocks. To access a data structure, participants of the application read and write blocks, but do not communicate with each other.

A DHT-based application faces four challenges. First, if a participant of the application crashes while modifying a DHT-based data structure, the data structure may be left in an inconsistent state. Second, because participants typically manipulate a shared data structure independently (i.e. without sending operations to a single server or server cluster), an application with concurrent participants also faces the challenge of providing consistency without direct use of serialization. Third, peer-to-peer systems are often used in situations where participants do not fully trust each other; thus another problem is how to defend against participants who maliciously damage the shared data structure. Fourth, DHTs typically replicate data in such a way that multiple partitions may have a complete copy of the data structure if a network outage occurs; thus applications using DHTs may experience conflicting updates in different partitions.

L^* is a set of techniques for maintaining consistent data structures on top of a DHT. L^* represents a data structure as a log of operations in the DHT, with a separate log per participant. That is, an application using L^* does not directly store the shared data structure in the DHT; instead, the data structure is implied by the history of operations in the logs, and L^* stores log records in the DHT. A participant updates the data structure by appending records to its log; a participant reads the current state of the data structure by scanning all participants' logs.

Logging allows participants to perform complex operations atomically with respect to participant failure. The use of a log for each participant mean that concurrent updates to the same data produce some acceptable outcome reflecting the operations, rather than a corrupted data structure. Logging operations also allow participants to undo un-wanted updates by a malicious participant retroactively. Finally, because each participant is likely to update its log within a single partition, after partitions merge, scanning the most recent logs naturally merges updates from different partitions.

PODC '03 Boston, Massachusetts, USA

Consistency

The heart of L^* is its algorithm for resolving the order of log records in different participants' logs. This algorithm deterministically produces a single ordering of log records. That is, L^* always chooses the same order for every two log records for all participants. This property means participants agree on the order of completed updates, even if those updates were issued concurrently. Participants can also use a mutual exclusion algorithm to ensure atomicity for multi-step operations. The algorithm works by appending log records to a participant's log and reading other logs. Chen et al [2] describe properties of L^* in more detail.

Experience

We built a multi-user peer-to-peer read-write file system, Ivy [1], that uses L^* to store all file system data and meta-data. The use of per-participant logs allows Ivy to support concurrent updates to the file system without using locks, and yet still maintain meta-data consistency. Ivy implements most file system operations without mutual exclusion; the only exceptions are file and directory creation. File and directory creation require mutual exclusion to avoid duplicate files or directories. Despite its use of logs, L^* makes it easy to build applications with good performance; Ivy caches aggressively, and checks the validity of the whole cache just by checking whether any logs have changed recently. A typical Ivy operation involves checking the set of logs (in parallel) for new log records, fetching new log records (if any), and then completing the operation entirely from the local cache.

References

[1] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A Read/Write Peer-to-peer File System. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*, Boston, Massachusetts. December, 2002.

[2] B. Chen, T. M. Gil, A. Muthitacharoen, and R. Morris. Building Data Structures on Untrusted Peer-to-Peer Storage with Per-participant Logs. MIT Laboratory for Computer Science TR 888. March, 2003.