Irminsule; a branch-consistent distributed library database

Thomas Gazagnaire (speaker), Amir Chaudhry, Jon Crowcroft,
Anil Madhavapeddy, Richard Mortier,[1] David Scott[2]
David Sheets and Gregory Tsipenyuk
University of Cambridge, University of Nottingham,[1] Citrix Systems[2]

1 Introducing Irminsule

Nowadays, typical applications spawn a large set of heterogeneous computing and storage devices with very different technical properties. For instance, the "cloud" is a high-latency, low bandwidth, high-availability storage device but can be costly; smart-phones and tablets are replaced every couple of years and are very limited in term of battery usage; set-top boxes are always on at home and can host plenty of data but are owned by your Internet provider.

Distributed applications running on these devices need to be able to choose between different kinds of policies:

Where to store what ? Applications should be able to choose to store only a subset of data in the cloud, with backup on a peer-to-peer network of set-top boxes

What level of trust ? Applications should be able to parameterize the level of trust against the storage device: for instance all data stored in the cloud should be encrypted, but not the temporary data stored in memory.

When to migrate data ? Applications should keep the different datastore in sync and they should be able to choose when to migrate data between them. For instance, smartphones should be synchronized with the "cloud" only when wifi is available.

How did that happen ? Monitoring and debugging of distributed applications is often a nightmare. We want applications (and users) to be able to easily inspect the history of changes with a clear explanation of the scheduling policies.

Instead of having a definitive answer to all these questions, Irminsule provides a collection of libraries for database primitives: base policies are available to the programmer, who can combine them to create a distributed application with persistent storage with complex scheduling. Irminsule stores can be fully compatible with the Git command-line (using a bi-directional translation) and is written in pure OCaml. Hence, it can run as a Mirage application on embedded devices or be compiled to JavaScript using `js_of_ocaml`.

Its source code is available under a BSD license: `https://github.com/samoht/irminsule`

2 Consistency Model

One of the most famous results related to distributed systems is the CAP theorem [5], which states that it is not possible to have a system which simultanously guarantees global consistency, high availability and resilience to network partition. The "modern" answer to this is to mantain availability but get rid of strong consistency [10, 6, 2]. This is also the approach taken by Irminsule – although strong consistency can be built on top of the existing substrate using consensus protocols if needed.

The weak form of consistency that we consider is inspired from distributed version controlled systems such as Git [11] and we dubbed it "branch" consistency: each device has its own (partial) replica which corresponds to a branch in the global database. Reads and writes are *local*, i.e. they happen only on the current branch, on data owned by the current replica.

Branches can then explicitly be merged together, at points in time controlled by the application, and using application-defined merge policies between replicas. Irminsule provides a library of base content implementations which satisfy the following signature:

```
module type CONTENTS = sig
  type t
  val merge: old:t -> t -> t -> t option
  ...
end
```

The library includes conflict-free replicated datatypes (CRDT) [9], custom-function and combinators to assemble these together. An interesting technical point: since we keep a complete history of changes, merging replicas having a common ancestor is much easier than with usual CRDTs where you usually need to build and keep track of vector clocks. For instance, mergeable counters can simply keep track of changes since they were forked:

```
type t = int
let merge ~old x y = old + (x-old) + (y-old)
```

3 Programming Model

We expose a mutable prefix-tree interface to the user, with support for local transactions, synchronization primitives across replicas, and snapshot/revert capabilities. The prefix tree `path` is usually a list of strings and node values are the user-defined mergeable `contents` (see 2):

```
module type S = sig
  type t
  type path
  type contents
  val read: t -> path -> contents
  val update: t -> path -> contents -> unit
  val remove: t -> path -> unit
  ...
end
```

The `S` signature also has the following properties:

Local Transactions Transactions are a useful tool to maintain some kind of sequential semantics, such as atomicity guarantees for a sequence of reads and writes. Their scope is limited to single replicas as argued in [1].

Synchronization A pub/sub [3] model similar to distributed version control systems with event notifications. The global "Agent" coordination policies are available as a library.

Snapshot/Revert Irminsule stores share the same property as any other Content-Addressed Store (CAS) where provenance tracking and snapshot/revert come for free.

Consistency Validation Using a technique similar to Merkle Tree [7] we can build proof of database consistency after a synchronization has taken place.

4 Heterogeneous Backends

The interface exposed by Irminsule is built upon two user-provided simple store signatures:

1. The *block store* is a low-level key-value append-only store, where values are a sequence of bytes and keys are deterministically computed from the values (for instance using `SHA` algorithms). The interface is very simple:

   ```
   module type BLOCK = sig
     type t
     type key
     val read: t -> key -> bytes
     val add: t -> bytes -> key
   end
   ```

   we provide a collection of base implementations parameterized over the hash and the value serialization functions (in-memory, Git, HTTP, bin-io) and we anticipate the community will contribute new backends as they must satisfy only a very minimal signature.

   The block store contains serialized values from both application contents, prefix-tree nodes and history meta-data. As there is no `remove` function, the store is expected to grow forever, but garbage-collection and compression techniques can be used to manage its growth. This is not an issue as commodity storage steadily becomes more and more inexpensive.

2. The *tag store* is the only mutable part of the system. This store is expected to be local to each replica, very small, and contains named pointers to keys in the block store (for instance, in a Git-like system, the `HEAD` tag points to the most recent commit object).

   ```
   module type TAG = sig
     type t
     type tag
     type key
     val read: t -> tag -> key
     val update: t -> tag -> key -> unit
     val remove: t -> tag -> unit
   end
   ```

The *high-level store* is automatically generated (as a functor application) over a block store, a tag store and the application contents description. It lifts immutable operations on the block store to a mutable interface to provide the interface discussed in 3.

```
module Make
  (B: BLOCK)
  (T: TAG with type key = BLOCK.key)
  (C: CONTENTS):
    S with type contents = C.t
```

The application can choose different block and tag store backends (for instance, the block store can serialized to Git and the tag store can be kept in memory).

5 Use-cases

We are co-developing Irminsule and different applications using it: The first one is a rewrite of *Xenstore* [4, 8], an efficient, single-host, in-memory database, with fast inter-process communication channels, which is critical to the proper functioning of hosts running the *Xen* hypervisor. In this application, Irminsule is being used to increase fault-tolerance and to provide a full diagnostic event tracing system.

We are also writing a versioned IMAP backend, using Irminsule to merge replicas instead of using the IMAP protocol. Finally, we are developing a new filesystem implementation using the branch consistency semantics, to explore and experiment with the (poorly specified) POSIX concurrent semantics.

References

[1] Sebastian Burckhardt, Daan Leijen, Manuel Fähndrich, and Mooly Sagiv. Eventually consistent transactions. In *Proceedings of the 21st European conference on Programming Languages and Systems*, ESOP'12, pages 67–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[2] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[3] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

[4] Thomas Gazagnaire and Vincent Hanquez. Oxenstored: An efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 203–214, New York, NY, USA, 2009. ACM.

[5] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[6] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[7] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378. Springer-Verlag, London, UK, UK, 1988.

[8] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group: Perspectives and perceptions. *SIGPLAN Not.*, 45(9):87–92, September 2010.

[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, Grenoble, France, October 2011. Springer.

[10] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[11] John Wiegley. Git from the bottom up, May 2008.