Paxos – a SWI-Prolog replicating key-value store

Jeffrey Rosenwald and Jan Wielemaker
CWI, Amsterdam
The Netherlands
E-mail: J.Wielemaker@cwi.nl

July 2, 2018

Abstract

This package provides the library `paxos.pl` that implements a replicating key-value store of Prolog terms on top of SWI-Prolog `broadcast` libraries and its TIPC or UDP based extension that allow broadcasting outside the process using networking.

Contents

1 paxos.pl: A Replicated Data Store

author Jeffrey Rosenwald (JeffRose@acm.org)
See also `tipc_broadcast.pl`, `udp_broadcast.pl`
license BSD-2

This module provides a replicated data store that is coordinated using a variation on Lamport's Paxos concensus protocol. The original method is described in his paper entitled, "The Part-time Parliament", which was published in 1998. The algorithm is tolerant of non-Byzantine failure. That is late or lost delivery or reply, but not senseless delivery or reply. The present algorithm takes advantage of the convenience offered by multicast to the quorum's membership, who can remain anonymous and who can come and go as they please without effecting Liveness or Safety properties.

Paxos' quorum is a set of one or more attentive members, whose processes respond to queries within some known time limit ($<$ 20ms), which includes roundtrip delivery delay. This property is easy to satisfy given that every coordinator is necessarily a member of the quorum as well, and a quorum of one is permitted. An inattentive member (e.g. one whose actions are late or lost) is deemed to be "not-present" for the purposes of the present transaction and consistency cannot be assured for that member. As long as there is at least one attentive member of the quorum, then persistence of the database is assured.

Each member maintains a ledger of terms along with information about when they were originally recorded. The member's ledger is deterministic. That is to say that there can only be one entry per functor/arity combination. No member will accept a new term proposal that has a line number that is equal-to or lower-than the one that is already recorded in the ledger.

Paxos is a three-phase protocol:

> 1: A coordinator first prepares the quorum for a new proposal by broadcasting a proposed term. The quorum responds by returning the last known line number for that functor/arity combination that is recorded in their respective ledgers.
>
> 2: The coordinator selects the highest line number it receives, increments it by one, and then asks the quorum to finally accept the new term with the new line number. The quorum checks their respective ledgers once again and if there is still no other ledger entry for that functor/arity combination that is equal-to or higher than the specified line, then each member records the term in the ledger at the specified line. The member indicates consent by returning the specified line number back to the coordinator. If consent is withheld by a member, then the member returns a `nack` instead. The coordinator requires unanimous consent. If it isn't achieved then the proposal fails and the coordinator must start over from the beginning.
>
> 3: Finally, the coordinator concludes the successful negotiation by broadcasting the agreement to the quorum in the form of a paxos(`changed(Key,Value)` event. This is the only event that should be of interest to user programs.

For practical reasons, we rely on the partially synchronous behavior (e.g. limited upper time bound for replies) of `broadcast_request/1` over TIPC to ensure Progress. Perhaps more importantly, we rely on the fact that the TIPC broadcast listener state machine guarantees the atomicity of `broadcast_request/1` at the process level, thus obviating the need for external mutual exclusion mechanisms.

Note that this algorithm does not guarantee the rightness of the value proposed. It only guarantees that if successful, the value proposed is identical for all attentive members of the quorum.

paxos_initialize(+*Options*) *[det]*

Initialize this Prolog process as a paxos node. The initialization requires an initialized and configured TIPC, UDP or other broadcast protocol. Calling this initialization may be omitted, in which case the equivant of `paxos_initialize([])` is executed lazily as part of the first paxos operation. Defined options:

node(*?NodeID*)

When instantiated, this node rejoins the network with the given node id. A fixed node idea should be used if the node is configured for persistency and causes the new node to receive updates for keys that have been created or modified since the node left the network. If *NodeID* is a variable it is unified with the discovered *NodeID*.

NodeID must be a small non-negative integer as these identifiers are used in bitmaps.

paxos_set(+*Term*) *[semidet]*

Equivalent to `paxos_key(Term,Key)`, `pasox_set(Key,Term)`. I.e., *Term* is a ground compound term and its key is the name/arity pair. This version provides compatibility with older versions of this library.

paxos_set(+*Key, +Value*) *[semidet]*
paxos_set(+*Key, +Value, +Options*) *[semidet]*

negotiates to have *Key*-*Value* recorded in the ledger for each of the quorum's members. This predicate succeeds if the quorum unanimously accepts the proposed term. If no such entry exists in the Paxon's ledger, then one is silently created. `paxos_set/1` will retry the transaction several times (default: 20) before failing. Failure is rare and is usually the result of a collision of two or more writers writing to the same term at precisely the same time. On failure, it may be useful to wait some random period of time, and then retry the transaction. By specifying a retry count of zero, `paxos_set/2` will succeed iff the first ballot succeeds.

On success, `paxos_set/1` will also broadcast the term paxos(`changed(Key,Value)`, to the quorum.

Options processed:

retry(*Retries*)

is a non-negative integer specifying the number of retries that will be performed before a set is abandoned. Defaults to the *setting* `max_sets` (20).

timeout(+*Seconds*)

Max time to wait for the forum to reply. Defaults to the *setting* `response_timeout` (0.020, 20ms).

Arguments

Term is a compound that may have unbound variables.

To be done If the *Value* is already current, should we simply do nothing?

paxos_get(*?Term***)** *[semidet]*

Equivalent to `paxos_key(Term,Key)`, `pasox_get(Key,Term)`. I.e., *Term* is a compound term and its key is the name/arity pair. This version provides compatibility with older versions of this library.

paxos_get(*+Key, +Value***)** *[semidet]*

paxos_get(*+Key, +Value, +Options***)** *[semidet]*

unifies Term with the entry retrieved from the Paxon's ledger. If no such entry exists in the member's local cache, then the quorum is asked to provide a value, which is verified for consistency. An implied `paxos_set/1` follows. This predicate succeeds if a term with the same functor and arity exists in the Paxon's ledger, and fails otherwise.

Options processed:

retry(*Retries***)**
: is a non-negative integer specifying the number of retries that will be performed before a set is abandoned. Defaults to the *setting* `max_gets` (5).

timeout(*+Seconds***)**
: Max time to wait for the forum to reply. Defaults to the *setting* `response_timeout` (0.020, 20ms).

Arguments

Term	is a compound. Any unbound variables are unified with those provided in the ledger entry.

paxos_replicate_key(*+Nodes:bitmap, ?Key, +Options***)** *[det]*

Replicate a *Key* to *Nodes*. If *Key* is unbound, a random key is selected.

timeout(*+Seconds***)**
: Max time to wait for the forum to reply. Defaults to the *setting* `response_timeout` (0.020, 20ms).

paxos_on_change(*?Term, :Goal***)** *[det]*

paxos_on_change(*?Key, ?Value, :Goal***)** *[det]*

executes the specified *Goal* when *Key* changes. `paxos_on_change/2` listens for paxos(`changed(Key,Value)` notifications for *Key*, which are emitted as the result of successful `paxos_set/3` transactions. When one is received for *Key*, then *Goal* is executed in a separate thread of execution.

Arguments

Term	is a compound, identical to that used for `paxos_get/1`.
Goal	is one of: • a callable atom or term, or • the atom `ignore`, which causes monitoring for *Term* to be discontinued.

Index