Beernet's API

Ruma R. Paul

November 24, 2015

1 Introduction

> **WARNING!** This document corresponds to the API of Beernet-1.0. However, Beernet-0.9 is still compatible with this document.

Beernet is provided as a Mozart-Oz [1] library. To use it, the program needs to import the main functor to create a peer to bootstrap a network, or to join an existing one using the reference of another peer. As any other ring *Structuted Overlay Network*, there is an identifier space, which a subset of $\mathbb{N}$. Each peer is associated with a unique Id from this identifier space: $[0..(2^n - 1)]$. Each Beernet network has a unique reference, RingRef, which is a record with pattern ring(name:<Atom> id:<Name>). The RingRef is set when the network is bootstrapped. Due to the unique id of each network, two independently bootstrapped Beernet networks are unable to merge by themselves (except for node join). In order to bootstrap a Beernet network, the first peer, which we will refer as Master, needs to be created. Importing the main functor and creating a Master peer works as follows:

```
functor
import
   Beernet at 'beernet/pbeer/Pbeer.ozf'
define
   Master = {Beernet.new args(firstAck:unit transactions:true)}
```

Importing the main functor and creating other peers, which will join the network bootstrapped by Master, works as follows:

```
functor
import
   Beernet at 'beernet/pbeer/Pbeer.ozf'
define
   Pbeer = {Beernet.new args(transactions:true)}
```

Each peer has a PeerRef, which is record with pattern pbeer(id:<Id> port:<Port>). Interacting with the peer is done by triggering an event as follows:

```
{Pbeer event(arg1 ... argn)}
```

We list now the different events that can be triggered on Beernet's peers. Even though Beernet's architecture is organized with layers where Trappist is the upper most one, the architecture does not prevent the access to lower layers because their functionality is important to implement applications.

2 Relaxed Ring

The following events can be used to get access to the functionality provided by the relaxed ring layer. It mostly provides access to peer's pointers and other information of the structured overlay network.

2.1 Basic Operations

- join(AccessRef) Triggers joining process using AccessRef as access point. AccessRef is a record with the pattern ref(pbeer:PeerRef ring:RingRef), where PeerRef is a peer's refernce, which is already on the network and RingRef is the reference of the Beernet network. A new peer can retrieve AccessRef by triggering getFullRef(?Res) event (described below) of an existing peer (which is already on the network).

- lookup(key:Key res:?Res) Triggers lookup for the responsible of Key, which will be passed through the hash function. Binds Res to the reference of the peer responsible of Key.

- lookupHash(HashKey) Triggers lookup for the responsible of HashKey without passing HashKey through the hash function. Binds Res to the reference of the peer responsible of Key.

- leave Roughly quit the network. No gently leave implemented.

2.2 Getting Information

- getId(?Res)
  Binds Res to the id of the peer.

- getRef(?Res)
  Binds Res to a record containing peer's reference with the pattern
  pbeer(id:<Id> port:<Port>).

- getRingRef(?Res)
  Binds Res to the ring reference of the Beernet network, where Res is bound to a record with pattern ring(name:<Atom> id:<Name>).

- getFullRef(?Res)
  Binds Res to a record containing peer's reference and ring's reference with the pattern ref(pbeer:<PbeerRef> ring:<RingRef>) .

- getMaxKey(?Res)
  Binds Res to the maximum key in ring's address space. Usually, the maximum key is $2^n - 1$, the default value of n is set as 21.

- getPred(?Res)
  Binds Res to the reference of peer's predecessor.

- getSucc(?Res)
  Binds Res to the reference of peer's successor.

- getRange(?Res)
  Binds Res to the responsibility range of peer with the pattern From#To, where From and To are integers keys.

- getKnowledge(?Res)
  Binds Res to the list of peer references accumulated in peer's *Knowledge Base*.

2.3 Other Events

- refreshFingers(?Flag) Triggers lookup for ideal keys of finger table to refresh the routing table. Binds Flag when all lookups are replied. Beernet supports *distributed k-ary search* as is generalized in [3]. The default value for k is set as 4.

- injectPermFail Peer stop answering any message, corresponds to failure of a peer.

- setLogger(Logger) Sets Logger as the default service to log information of the peer. Mostly used for testing and debugging.

- introduce(PeerRef) Introduces a peer to the current peer by including PeerRef to peer's *Knowledge Base*, which is a list of all PeerRefs, known to the peer. Can be used as an *Oracle* to give information to the system.

3 Message Sending

This section describes the events that allow applications to send and receive messages to other peers.

- send(Msg to:Key)
  Sends message Msg to the responsible of key Key. It uses Beeret's routing to find the peer responsible for key Key.

- dsend(Msg to:PeerRef)
  Sends a direct message Msg to a peer using PeerRef, thus does not need to use Beernet's routing.

- receive(?Msg)
  Binds Msg to the next message received by the peer, and that it has not been handled by any of Beernet's layer. It blocks until next message is received. Multiple receive on the same peer will each read the full stream of messages received by the peer. This allows modularity in programming.

4 DHT

Beernet also provides the basic operations of a distributed hash table (DHT). None of this uses replication; therefore, there are no guarantees about persistence.

- put(Key Val)
  Stores the value Val associated with key Key, only in the peer responsible for the key resulting from applying the hash function to key Key.

- get(Key ?Val)
  Binds Val to the value stored with key Key. It is bound to the atom `'NOT_FOUND'` in case that no value is associated with such key.

- delete(Key)
  Deletes the item associated to key Key. *WARNING!* This operation is very dangerous, as this will re-initialize the version number of the Key at the peer. This may create conflict with other replicas of key Key during a transaction.

5 Symmetric Replication

Beernet uses *Symmetric Replication* [4], where the replicas of each key are symmetrically placed across the network using a regular polygon of f sides, where f is the replication factor. The symmetric replication layer does not provide an interface to store values with replication, but it does provides some functions to retrieve replicate data, and to send messages to replica-sets.

- getFactor(?Res)
  Binds Res to the replication factor of each key. The default value is set as 4.

- setFactor(NewReplicationFactor)
  Sets replication factor of the peer as NewReplicationFactor.

- bulk(Msg to:Key)
  Sends message Msg to the replication set associated to key Key. Mainly used by the functions to retrieve replicate data and the *Trappist* layer to commit modified value of a key Key during a transaction, at the majority of the replicas.

- findRSet(?Flag)
  Finds and collects references of other members of the replication set associated to peer's id. Binds Flag when replies from all other members of the replication set have been received.

- getOne(Key ?Val)
  Binds Val to the first answer received from any of the replicas of the item associated with key Key. If value is `'NOT_FOUND'`, the peer does not bind Val until it gets a valid value, or until all replicas has replied `'NOT_FOUND'`.

- getAll(Key ?Val)
  Binds Val to a list containing all values stored in the replica set associated to key Key.

- getMajority(Key ?Val)
  Binds Val to a list containing the values from the replica set associated to key Key. It binds Val as soon as the majority is reached.

6 Trappist

Trappist is able to support different protocols to run transactions on replicated items. However, for current version of Beernet, only *Paxos Consensus* protocol is improved and verified. *Snapshot Isolation* is adopted in Beernet's transaction layer, which guarantees that all reads made in a transaction reads the last committed values that existed at the time it started and transaction itself will successfully commit only if no updates it has made conflict with any concurrent updates made since that snapshot.

6.1 Paxos Consensus

- runTransaction(Trans Client Protocol)
  Run the transaction Trans using protocol Protocol. The answer, commit or abort is sent to the port Client. Currently, the protocol supported by this interface is only paxos.

- executeTransaction(Trans Client Protocol)
  Exactly the same as runTransaction. Kept only for backward compatibility.

Inside a transaction, there are three operations that can be used to manipulate data.

- write(Key Val)
  Write value Val using key Key. The new value is stored at least in the majority of the replicas. Updating the value gives a new version number to the item.

- read(Key ?Val)
  Binds Val to the latest value associated to key Key. Strong consistency is guaranteed by reading from the majority of the replicas.

- remove(Key)
  Removes the item associated to key Key from the majority of the replicas. *WARNING!* This operation is very dangerous, as this will re-initialize the version number of the Key at the peer. It is suggested to do it at the end during data cleanup.

6.2 Paxos with Eager Locking

- getLocks(Keys ?LockId)
  Get the locks of the majority of replicas of all items associated to the list of keys Keys. Binds LockId to TMPeerId#TId#TMId#LockPeriod if locks are successfully granted, and to error otherwise.

  - TMPeerId is the Id of the peer who is the coordinator of this transaction.
  - TId is an Oz name, which is the Id of this transaction.
  - TMId is an Oz name, which is the Id of the transaction manager, i.e., coordinator of this transaction.
  - LockPeriod is an Integer, which the time duration (in milliseconds) for which the lock has been granted.

- commitTransaction(LockId KeyValuePairs) update all items of the list KeyValuePairs which must be locked using LockId. Each element of the list KeyValuePairs must be of the form <key>#<value>

- Two notifications can be received using the receive event described previously, in context with a transaction using Paxos with Eager Locking.

  - lockExpire(LockId) to notify the client about the expiration (after the LockPeriod, described before, is over) of previously granted locks using LockId.
  - lockChange(OldLOckId NewLockId) to notify the client about the change of LockId. This notification will be issued when the TM of a transaction fails, in which case a RTM takes over the responsibility of the TM.

6.3 Notification Layer

- becomeReader(Key)
  Subscribes the current peer to be notified about locking and updates of the item associated to key Key. Notification are received using the receive event described previously.

7 Phase Information

A peer provides information to the application layer regarding its current qualitative phase. The phase of each peer is directly co-related with the functionalities currently available through the peer. We have identified three clearly distinguishable phases of a peer: *Solid*, *Liquid* and *Gaseous*, where the liquid phase of a peer consists of three immiscible sub-phases. We define semantics of each phase and sub-phases.

- solid
  If the peer has stable predecessor and successor pointers (i.e., the peer is on core ring), along with a stable finger table. It can be safely assumed that such peer can support efficient routing, thus accommodate up-to-date replica sets, thus leading to all the upper layer functionalities.

- liquid
  If the peer is on a branch, it is less strongly connected than the solid phase. However a peer can be on a branch temporarily, e.g., as part of the join protocol or due to false suspicion as a result of sudden slow-down of underlying physical link. We identify three immiscible liquid sub-phases.

  - liquid-1
    If the peer is on a branch, but the depth of the peer (distance from the core ring) is less than or equal to 2. Also, the peer still holds a stable finger table. The justification of depth of 2 for this sub-phase is based on the evaluation of average branch sizes in [2], where it is shown that the average size of branches of Beernet is ≤ 2, corresponding to the connectivity among peers on the Internet. So, if a peer's depth on a branch is ≤ 2, the operating environment from a peer's perspective is still the usual one, it might temporarily be pushed on a branch. From the application's perspective, the peer is still able to provide all the higher layer functionalities.
  - liquid-2
    If the peer is on a branch, but the depth of the peer (distance from the core ring) is greater than 2 and it is not the tail of the branch. Also, the finger table at the peer still holds $> 50\%$ valid fingers. So, the peer is still able to support at least all DHT operations.

  - liquid−3
    If the peer is on a branch with a depth > 2 and it is the tail of a branch. As discussed in [2], the tail of a branch has higher probability to get isolated during churn, thus introducing unavailability in the key range. Also, most of the fingers in the peer's finger table are invalid or crashed. From the application perspective, the peer in this sub-phase provides very limited functionality, mostly basic connectivity through its successor pointer.

- gas
  If the peer is isolated, i.e., no connection with any other peers of the system.

7.1 Notification Layer

The notification layer supports both push and pull method of injecting phase information of the peer to the application layer.

- getPhase(?Res)
  Binds Res to the current phase of the peer, where Res will have one of the phase defined above.

- setPhaseNotify
  Sets PhasePush flag of the peer. The peer triggers periodic checking of its phase, if a phase transition occurs, it sends notification using the receive event described previously.

References

[1] Mozart Consortium. The Mozart-Oz programming system. http://mozart.github.io/, 2013.

[2] Boris Mejias Candia. Beernet: A Relaxed Approach to the Design of Scalable Systems with Self-Managing Behaviour and Transactional Robust Storage. PhD thesis, ICTEAM, Universite catholique de Louvain, Louvain-la-Neuve, Belgium, October 2010.

[3] Luc Onana Alima, Sameh El-ansary, Per Brand, and Seif Haridi. Dks(n,k,f): a family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid), pages 344350, 2003.

[4] Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables. PhD thesis, KTH Royal Institute of Technology, Sweden, 2006.