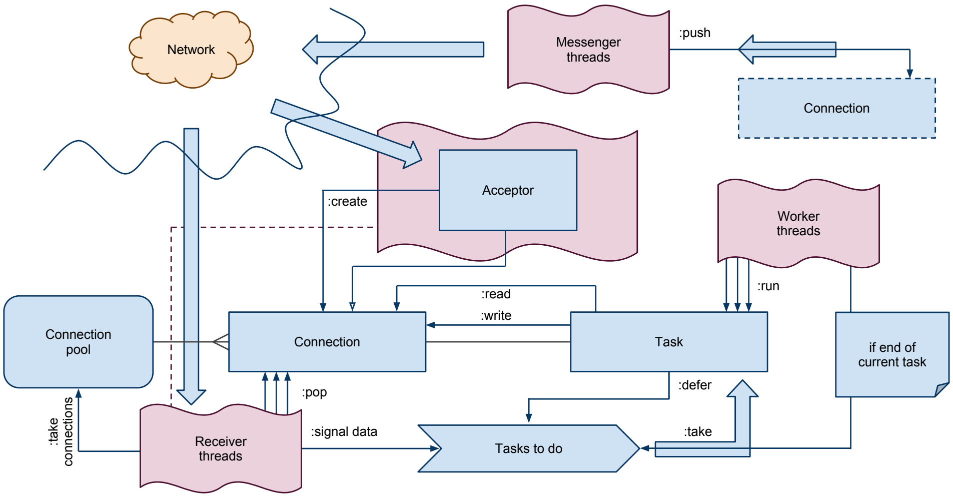
CoherentDB Net server project

Michał Stachurski Rafał Rawicki Cezary Bartoszuk

November 15, 2010



1 Threads

The basic description of threads working in the system.

1.1 Reveiver thread

```
ReceiverThread:
    loop:
        active_connections := ConnectionPool.receive_connections(self)
        for active_connection in active_connections:
            active_connection.pull()
```

Acceptor is also run at a *receiver thread* all the magic is hidden in the `pull` method of the connection object. Described below.

1.2 Worker thread

Supposing *Tasks to do* queue is a monitor the pseudo-code looks similar to this:

```
WorkerThread:
    loop:
        task := tasks_to_do.pop()
        task()
```

1.3 Messenger thread

Similar to *Worker thread*, but gets data from *messages* queue.

```
MessengerThread:
    loop:
        active_connections := ConnectionPool.send_connections(self)
        for active_connection in active_connections:
            active_connection.push()
```

The data burst mechanism will be probably hidden in the connection's `push` implementation.

2 Queues

Both *Tasks to do* and *Messages* are monitor-synchronized queues.

2.1 Tasks to do queue

A FIFO (or maybe Priority?) queue. A monitor with standard producer-consumer idiom:

```
monitor tasksToDo:
    var buffer: Queue
    var size: Integer
    var full: Condition
    var empty: Condition
    method push(item):
        while size = MAX_SIZE:
            full.wait
        buffer.push(item)
        size := size + 1
        if size = 1:
            empty.notify
    method pop:
        while size = 0:
            empty.wait
        item := buffer.pop
        size := size − 1
        if size = MAX_SIZE − 1:
            full.notify
        return item
```

The C++-specific implementation will probably be parameterizable with a queue (so that any data structure with `pop` and `push` - or similar - operations will be apropriate).

2.2 Messages queue

Probably the same idea as *Tasks to do* queue - e.i. producer-consumer pattern with parameterizable queue-like buffer inside.

3 System-wide objects

One object, exactly...

3.1 Connection pool

Represents a set of `Connections` (including the `Acceptor`).

- Behaviour
  - `receive_connections(thread: Thread)`
    Gives a handle to a set of connections on which data is waiting (e.g. performs a form of `poll` or `select`).
  - `send_connections(thread: Thread)`
    Returns handle to a set of connections which are send-ready (e.g. have messages pending and sufficient quota).

  - `new_connection()`
    Creates a new connection in the pool.

4 Per-session objects

4.1 Connection

Represents a network connection.

- Fields
  - `fd: FileDescriptor`
  - `read_observers: Queue<Size * Callback(Data) * Time>`
  - `outgoind_messages: Queue<Data>`
  - `read_buffer: Buffer`
  - `read_buffer_size: Size`
- Behaviour
  - `read`
    Arguments:
    * `nbytes: Size` requested data length
    * `callback: Data -> Void` function called when data comes
    * `timeout: TimeDelta` read kills the session if data won't come before time specified here passes

    Registers a read observer. Guarantees calling the `callback` function with bound `data` argument (which represents read data).
  - `write`
    Arguments:
    * `bytes: Data` data we want to send
    * `timeout: TimeDelta` kill the connection if the server won't be able to send the data before time specified here passes.

    Adds `bytes` to the messages-to-send queue (which is connected with the connection).
  - `read_timeout`
    Arguments:
    * `timeout: TimeDelta`

    Kills the connection if `read` isn't called by the time specified in `timeout`.
  - `write_timeout`
    Arguments:
    * `timeout: TimeDelta`

    Kills the connection if `write` isn't called by the time specified in `timeout`.

  - `pull`
    If an observer waits for data on this connection (via calling `read` before) and data is waiting then `pull` caches the incoming data. If apropriate amount of data is buffered the first action in waiting-for-read observers queue is pushed to the tasks queue.

    `Acceptor` as a special kind of `Connection` implements the `push` method which creates a new connection (runs `accept` under the hood).
  - `push`
    Tries to send data from the messages-to-send queue.

4.2 Task

An object representing a callable.
Actually a non-argument procedure, e.i. `function: Void -> Void`.

4.3 Join point

Provides mechanism for starting computation after specified amount of threads end.

- Fields
  - `counter: UInteger`
  - `callback: Void -> Void`
  - `const start_counter: UInteger`
- Behaviour
  - `lazy_join`
    Decreases `counter`. If it reaches 0 then it `defer`s the `callback`.
  - `join`
    Same as `lazy_join` but instead of `defer`ing an action - runs it immidiately.
  - `init`
    Arguments:
    * `nthreads: Size`
    * `callback: Void -> Void`

    Sets the `callback` and the amount of threads that would join on this join point.

4.4 Defer

`defer(action: Void -> Void)`
Pushes given `action` to the tasks queue.